1988 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING
Vol. II Software Sturgis

PENN STATE

INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

PROCEEDINGS OF THE 1988 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

August 15-19, 1988

Vol. II Software

Howard E. Sturgis, Editor

Sponsored by

THE PENNSYLVANIA STATE UNIVERSITY · 1855 ·

Department of Electrical Engineering
PENN STATE UNIVERSITY
University Park, Pennsylvania

THE PENNSYLVANIA STATE UNIVERSITY PRESS
UNIVERSITY PARK AND LONDON

PROCEEDINGS

OF THE

1988 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

August 15-19, 1988

Vol. II Software

Howard E. Sturgis, Editor

Sponsored by

Department of Electrical Engineering
PENN STATE UNIVERSITY
University Park, Pennsylvania

THE PENNSYLVANIA STATE UNIVERSITY PRESS
UNIVERSITY PARK AND LONDON

Library of Congress Catalog Card Number 79-640377
ISSN 0190-3918
ISBN 0-271-00654-4
IEEE Computer Society Order Number 889
IEEE Catalog Number 88CH2625-2

Additional copies may be obtained from:
Penn State Press
215 Wagner Building
University Park, PA 16802

PREFACE

Interest in the field of parallel processing continues to climb. This trend is evidenced by the sharp increase in papers submitted to the International Conference on Parallel Processing during recent years:

Year	Papers Submitted	Papers Accepted	Percent
1980	170	65	57
1983	240	136	57
1986	400	170	43
1987	487	174	36
1988	590	173	29

Although the number of submissions continues to increase, the number of accepted papers this year and in the past two years has remained relatively unchanged. This is due to the limitation imposed by the fixed number of hours available for the conference. As a result, a record number of papers had to be rejected. This year, the conference proceedings is being published in three volumes according to the subject category. The breakdown of submissions and acceptances in the three main categories of this conference is as follows:

Category	Papers Submitted	Papers Accepted	Percent
Architecture	264	74	28
Software	144	43	30
Algorithms and Applications	182	56	31

Of the 173 papers that were accepted, 79 were accepted as regular papers and 94 were accepted as short papers. Many papers that normally would have been accepted as long papers were accepted as short papers in order to meet the maximum number of paper-sessions allotted for the conference.

Finding sufficient numbers of qualified reviewers was a particularly challenging task this year, due to the record number of submissions. Over 1,000 professionals in the field participated in this process. This year the process of selecting referees was simplified by the use of questionnaires, which were mailed to previous participants in the conference. The information on the completed questionnaires was entered into databases, which then allowed the conference chairmen to select reviewers qualified in fairly specialized fields. Even so, numerous papers were so highly specialized that custom selection of referees was still required. It appears that an even more detailed breakdown of specializations will be needed for these questionnaires in the future. Greater effort will also be required in the future to find additional reviewers to match the increasing numbers of submissions.

I wish to thank the management of the Xerox Palo Alto Research Center for providing me the opportunity to co-chair the ICPP88 program. I also thank Prof. Tse-yun Feng, my immediate supervisors Dr. John White and Dr. Robert Ritchie, and my colleagues in the Computer Science Laboratory for their support and encouragement. Finally, I offer special thanks to my Secretary Kathi Anderson; this project would have been impossible without her assistance.

Howard E. Sturgis
Program Co-chair
Xerox Palo Alto Research Center
Palo Alto, California 94304

LIST OF REFEREES

Abdollahzadeh, F.	U. of Toledo
Agre, J.	Rockwell
Akingbehin, K.	U. of Michigan
Amit, N.	U. of Minnesota
Appelbe, W.	Georgia Inst. of Tech.
Arafeh, B.R.	Auburn U.
Atwood, J.W.	Concordia U.
Azimi, M.	Michigan State U.
Baden, S.	Lawrence Berkeley Lab.
Badii, M.	U. of Toledo
Baer, J-L.	U. of Washington
Bagrodia, R.L.	UCLA
Bailey, D.A.	U. of Massachusetts
Bailor, P.D.	Air Force Inst. of Tech.
Baldwin, D.	U. of Rochester
Bannister, J.	The Aerospace Corp.
Bansal, A.K.	Case Western Reserve U.
Barkhordarian, S.P.	EIP Microwave, Inc.
Baumgartner, K.M.	Coordinated Science Lab.
Beck, R.D.	Sequent Computer Systems
Bennet, T.W.	U. of Maryland
Benson, M.	U. of Minnesota Duluth
Bernstein, D.	T.J. Watson Rsch. Cntr.
Bobbie, P.O.	U. of West Florida
Bose, P.	T.J. Watson Rsch. Cntr.
Bradley, D.K.	U. of Illinois-Urbana
Browne, J.C.	U. of Texas
Bryant, B.R.	U. of Alabama-Birmingham
Burdorf, C.	Rand Corp.
Busse, T.	Computer Sciences Corp.
Callahan, D.	Rice U.
Canas, D.A.	Wake Forest U.
Cann, D.C.	Colorado State U.
Carlson, W.W.	Purdue U.
Casavant, T.	Purdue U.
Chan, E.Y.	Harris Corp.
Chang, C.K.	U. of Illinois-Chicago
Chen, A.C.	AT&T Information Systems
Chen, K-W.H.	AT&T Information Systems
Christopher, T.W.	Illinois Inst. of Tech.
Copper III, A.N.	Bowling Green State U.
Crockett, T.W.	NASA Langley Rsch. Cntr.
Cytron, R.	T.J. Watson Rsch. Cntr.
Daffinger, C.A.	Indiana U.
Danforth, S.	MCC
Dekel, E.	U. of Texas-Dallas
Demers, A.	Xerox PARC
Dinning, A.	Courant Inst.
Ellis, C.S.	Duke U.
Erdogan, S.S.	San Diego State U.
Eventoff, W.	Concurrent Computer Corp.
Fang, Z.	Concurrent Computer Corp.
Finkel, R.A.	U. of Kentucky
Fisher, A.L.	Carnegie Mellon U.
Flynn, S.F.	Courant Inst.
Forgy, C.	Prod. Systems Technologies
Forin, A.	Carnegie-Mellon U.
Forrest, S.	Teknowledge Inc.
Fortes, J.A.B.	Purdue U.
Fowler, R.	U. of Rochester
French, J.C.	Inst. for Parallel Computation
Frey, M.G.	FERMILAB
Fujimoto, R.M.	U. of Utah
Gafter, N.M.	U. of Rochester
Gantenbein, R.E.	U. of Wyoming
Gehani, N.	AT&T Bell Labs
Ghafoor, A.	Syracuse U.
Gokhale, M.B.	U. of Delaware
Goldman, R.	Lucid, Inc.
Gooley, M.M.	U. of Illinois-Urbana
Grit, D.H.	Colorado State U.
Grogono, P.	Concordia U.
Guarna, Jr., V.A.	U. of Illinois-Urbana
Gunderson, A.S.	GTE Labs Inc.
Gupta, R.	Philips Labs
Guthrie, G.	Maharishi International U.
Guzzi, M.D.	U. of Illinois-Urbana
Hailperin, M.	Knowledge Systems Lab.
Hawkinson, S.W.	Supercomputer Systems, Inc.
Headington, M.R.	U. of Wisconsin-LaCrosse
Hemmendinger, D.	Wright State U.
Herlihy, M.	Carnegie Mellon U.
Hoeflinger, J.	U. of Illinois-Urbana
Hoppe, A.	Louisiana State U.
Horvath, J.C.	JPL
Hsieh, T.S.	The Aerospace Corp.
Huang, C.H.	Northwestern U.
Huang, T.L.	Northwestern U.
Husmann, H.E.	AT&T Bell Labs
Hwang, K.D.	U. of Illinois-Urbana
Janicki, R.	McMaster U.
Juang, J-Y.	Northwestern U.
Kahn, K.	Xerox PARC
Kale, L.V.	U. of Illinois
Kandt, R.K.	Teknowledge Federal Systems
Katz, M.	Rockwell International Corp.
Kesselman, C.	The Aerospace Corp.
Khoshafian, S.	Ashton-Tate

Kim, D.W.	U. of Texas-Austin	Pramanik, S.	Michigan State U.
Knox, D.L.	U. of Wisconsin-Milwaukee	Preiss, B.R.	U. of Waterloo
Koegel, J.F.	U. of Denver	Quinn, M.J.	U. of New Hampshire
Koelbel, C.	Purdue U.	Quiroz, C.A.	U. of Rochester
Kogge, P.M.	IBM	Ramanujan, R.S.	Honeywell
Krauss, K.G.	Easton, PA	Rana, S.P.	Wayne State U.
Lastra, A.A.	Duke U.	Ravi, T.M.	U. of California
Lato, K.A.	AT&T Bell Labs	Rego, V.	Purdue U.
Lauer, P.E.	McMaster U.	Reinhardt, S.	Cray Research, Inc.
LeBlanc, T.	U. of Rochester	Riccardi, G.A.	Florida State U.
Lee, G.	U. of SW Louisiana	Rickert, N.W.	Northern Illinois U.
Lee, J.	Rockwell International Corp.	Roberts, E.S.	Digital Equipment Corp.
Lee, P-N.	U. of Houston	Salfi, R.E.	Summit, NJ
Levy, H.	U. of Washington	Saphier, S.H.	Dept. of Defense
Li, K.	Princeton U.	Sarkar, V.	IBM
Li, P.P.	AMETEK	Schacht, E.N.	Computer Sciences Corp.
Li, Z.	U. of Illinois	Schwetman, H.	Tech. Corp.
Lin, K-J.	U. of Illinois-Urbana	Scott, M.L.	U. of Rochester
Liu, L.Y.	Penn. State	Seyedoff, H.	U. of Oklahoma
Lo, V.M.	U. of Oregon	Sheets, K.B.	AT&T
Loganantharaj, R.	U. of SW Louisiana	Shepard, T.	Royal Military College
Loucks, W.M.	U. of Waterloo	Sheu, P.C-Y.	Purdue U.
Mace, M.	Duke	Shi, Y.	Temple U.
Madison, D.E.	Naval Postgraduate School	Shirazi, B.	Southern Methodist U.
Malony, A.D.	U. of Illinois-Urbana	Shu, W.W.	U. of Illinois-Urbana
Manwaring, M.L.	Washington State U.	Simpson, R.T.	Encore Computer Corp.
Marshall, R.C.	IBM Corp.	Singh, V.	Stanford U.
Marti, J.B.	The Rand Corp.	Singhal, M.	Ohio State U.
Mauney, J.	N. Carolina State U.	Skedzielewski, S.K.	Livermore, CA
McClosky, M.J.	San Diego, CA	Smith, K.	Georgia Inst. of Tech.
McKinley, P.K.	U. of Illinois-Urbana	Smith, L.L.	Fort George Meade
Mercer, R.W.	Convex Computer Corp.	Smith-Thomas, B.	AT&T Bell Labs
Middleton, D.	NASA Langley Rsch. Cntr.	Socha, D.	U. of Washington
Mills, P.H.	U. of North Carolina	Sridharan, K.	Concurrent Computer Corp.
Miranker, D.P.	U. of Texas-Austin	Stone, J.M.	T.J. Watson Rsch. Cntr.
Miyashita, J.	California State U.	Stotts, P.D.J.	U. of Maryland
Mullin, L.R.	Syracuse U.	Strout II, R.E.	Microsystems, Inc.
Musciano, A.J.	Harris Corp.	Subramanian, R.	AT&T Bell Labs
Natarajan, K.S.	T.J. Watson Rsch. Cntr.	Szymanski, B.K.	Reusselaer Polytechnic Inst.
Natour, I.A.	Western Michigan U.	Tai, H-M.	Univ. of Tulsa
Nikhil, R.	MIT	Tal, D.	Florida International U.
Notkin, D.	U. of Washington	Tang, P.	U. of Illinois
Oldehoeft, A.E.	Iowa State U.	Tanik, M.M.	Southern Methodist U.
Ottenstein, K.J.	BBN Advanced Computers, Inc.	Tenny, L.	Indiana U.
Pagan, M.	RCA Advanced Tech. Labs	Thomas, R.H.	BBN Advanced Computers, Inc.
Palis, M.A.	U. of Pennsylvania	Tinker, P.	Rockwell
Pan, Z.	New Mexico State U.	Tomboulian, S.	NASA Langley Rsch. Cntr.
Pickert, J.	U. of Illinois-Urbana	Tong, Z.	U. of Minnesota
Pieper, K.L.	Stanford U.	Toomey, L.	IBM
Plishka, R.M.	U. of Scranton	Tripathi, A.	U. of Minnesota
Potter, J.L.	Lanham, MD	Tsai, J.P.	U. of Illinois-Chicago
Prabhu, G.M.	Iowa State U.	Tyan, H.R.	Northwestern U.

Tyrer, H.W.	U. of Missouri-Columbia	Weiser, M.	Xerox PARC
Vishnubhotla, P.	The Ohio State U.	Weiss, M.	COMPASS
Vosbury, N.	Unisys Corp.	Wikstrom, M.	Iowa State U.
Wah, B.W.	NSF	Wolfe, M.	Kuck & Associates, Inc.
Wang, C-C.	T.J. Watson Rsch. Cntr.	Wright, C.	Iowa State U.
Wang, P.	George Mason U.	Wu, M-Y.	U. of California-Irvine
Weinstock, C.	Software Engineering Inst.	Xu, Z.	Rutgers U.

AUTHOR INDEX

TABLE OF CONTENTS

Parallelism in Connection-Graph-Based Logic Inference

Jie-Yong Juang and Ting-Lu Huang
Dept. of Electrical Engineering & Computer Science
Northwestern University
Evanston, Illinois 60208
Tel: (312) 491-7103

Ed Freeman
U S West Advanced Technologies
6200 South Quebec Street
Englewood, Colorado 80111
Tel: (303) 889-6036

Abstract:
In this paper, we investigate the parallelism that can be achieved by concurrent resolution on a predicate connection graph. Predicate connection graphs provide a sound basis for parallel logic inference. However, unrestricted concurrent resolution on this graph may lead to logical inconsistency. This seems to contradict the completeness theorem of the resolution principle and the common belief of independence in parallel resolution. Using Bernstein conditions, logical inconsistency is found to be a problem of out-of-sequence manipulations of the connection graph. Thus, to prevent logical inconsistency, two resolutions must be executed in a sequential order if there is an overlap in the parts of graph they manipulate. Only limited parallelism is possible in this case due to the large extent of snowball effect. Fortunately, we have shown that precedence constraints between two resolutions can be relaxed. This property allows the proposed *lock-and-withdraw* synchronization scheme to exploit a high parallelism at its level of abstraction. To reduce synchronization overhead, we have also proposed a graph partitioning approach. In this approach, synchronization is necessary only when a resolution involves boundary clauses. Since each subgraph of a partition can be distributed among memory banks evenly, memory conflicts can also be minimized using the partitioning approach. For message-based multiprocessors such as the hypercube, we suggest that graph updates across the partition boundary be stored and re-constructed when each part of the subgraph is to be used. This approach can further minimize time-consuming message-based synchronization.

1. Introduction

Resolution has been the basis of logic inference since its first introduction in 1965 [13]. However, its execution on today's computers is too slow to be effective, primarily due to the long resolution cycle time and to exponential complexity. Although exponential explosion remains unavoidable, connection-graph-based resolution procedures have been shown to be a promising solution [3, 8, 14]. Such procedures organize the input clauses of a problem formulation into a predicate connection graph [9] which offers several distinct advantages over previous approaches. First, once the connection graph is constructed all information regarding resolvable literals is maintained, and therefore no further searching for unifiable clauses is needed. Second, a link is deleted after it is resolved. Its parent clauses and associated links can also be deleted if the removal of the link results in a pure literal. Such deletions can continue for all the adjacent clauses, and leads to a snowball effect that causes a rapid reduction of the graph. Third, unlike AND/OR-tree-based inference procedures [10] in which the search space is built up gradually as the inference proceeds, the whole search space of a connection-graph-based procedure is known. This can better facilitate the implementation of various problem-solving strategies such as subsumption [7], paramodulation [16], etc. The presence of the complete search space also provides a sound basis for parallel logic inference since task distribution can be done more effectively.

This material is based upon research supported by
U S West Advanced Technologies, Englewood, Colorado

Rapid deletion of unnecessary clauses and links helps keep a connection graph concise. As a result, non-determinism is reduced, useless resolutions [17] can be minimized, and the procedure is less likely to explode. Nevertheless, the deletion may cause *logical inconsistency* when links are resolved concurrently without careful coordination. Logical inconsistency is a problem in which an *unsatisfiable* clause set is falsely changed to a *satisfiable* one [4]. Consequently, subsequent resolutions will not lead to an empty clause, and the inference will fail to find the correct answer (as it would be able to if the logical inconsistency had not occurred). Thus, proper synchronization is necessary in connection-graph-based parallel logic inference procedures.

In this paper, we examine the parallelism achievable in connection-graph-based inference procedures according to Bernstein conditions. This allows us to design better connection-graph-based parallel logic inference procedures for different architectures using different synchronization mechanisms. A review of connection graph resolution procedure is given in Section 2. We will then describe the logical consistency problem followed by a summary of solutions found in the literature. In Section 4, Bernstein conditions are applied to explain why logical inconsistency occurs. We then describe how parallelism can be fully exploited under different types of conditions. Finally, we describe how one might design more efficient and practical connection-graph-based parallel logic inference procedures.

2. Logic Inference Based on Connection Graphs

A predicate connection graph of an input clause set can be constructed as follows: each literal of a clause in the input clause set is represented by a node in the graph, and the nodes representing literals of a clause are grouped together. Unification is then conducted to match every pair of literals which have the same predicate symbol and are complementary in sign. If the unification attempt between two literals has succeeded, then the two corresponding nodes are marked by a link and the resulting MGU (the most general unifier) is used to label the link. Given the clause set of Figure 9(a) in Section 6, its corresponding connection graph is shown in Figure 9(b). After the connection graph is constructed, a resolution procedure then repeatedly selects a link, resolves upon it, generates the associated resolvent, and finally inserts this resolvent into the connection graph. This process repeats until a null resolvent is generated or until further resolution is impossible.

Each resolvent inherits the unifiable links from its two parent clauses, and the new MGUs of these links are obtained by the composition of the old MGU and the MGU used in the current resolution. Substitution compatibility is checked and incompatible links are not inherited. After the resolvent and its links are generated, the link previously used to conduct the resolution is removed from the two parent clauses.

If the resolvent is not an empty clause, it is checked for deletion due to *tautology* or *pure literals.* Because tautologies do not positively contribute to the inference, they can be deleted from a set of clauses without affecting the *unsatisfiability of refutation.* In connection graph, a literal becomes pure when it does not have any link incident to it (i.e., it is an isolated node). A clause containing a pure literal can not contribute to a refutation because the unlinked literal can never be resolved upon [8, 13]. Either of the parent clause can become pure after the removal of the resolved link. These clauses are subsequently deleted from the connection graph.

Deletion of clauses containing pure literals is an important feature of the connection graph resolution procedure. In addition to the clause itself, all links connected to its literals must also be deleted from the graph. Deletion of such links, however, may cause literals in other clauses to become disconnected. Thus deletion of clauses can create a *snowball effect* such that a succession of clauses is deleted from the graph. Deletion of clauses simplifies the connection graph, reduces the search space, and makes it easier to find a solution.

3. Examples of Logical Inconsistency

Sequential resolution upon links in a predicate connection graph has been shown to be sound and consistent [8, 14, 15]. Nevertheless, as identified in the literature, parallel resolutions on such a graph often result in logical inconsistency [5, 11]. That is, during a parallel inference, these procedures may change an unsatisfiable clause set into a satisfiable one. As a result, subsequent resolution lead to an empty set instead of to an empty clause. No answer can be derived from the result. This contradicts the soundness and completeness of the resolution principle [4]. To provide better insight into the problem, we illustrate how it can occur using the following examples.

3.1 Example 1

A connection graph of four clauses is shown in Figure 1. Resolutions are performed upon the graph by two processors concurrently. Processor 1 is resolving on Link 1, and Processor 2 is resolving on Link 2. After the resolvents e and f are generated, the two resolved links are removed from the graph, and both processors try to establish links for the new clauses. Processor 1 finds no link to be inherited from Clause a. It then consults Clause b, and sees a single link incident to Clause b, i.e., Link 6. So, it connects Clause e to Clause c via a new link (Link 4). Since the two processors run concurrently, the new link may be established after Processor 2 has consulted Clause c. In this case, Processor 2 will be aware the existence of the new link, and Clause f is thus connected to Clause b only. Should Processor 2 have examined Clause c after Processor 1 had connected its resolvent to Clause c, Clause f would have inherited two links from c, namely Links 3 and 5.

Link 5 plays an important role in the subsequent resolutions. Removing Links 1 and 2 leaves literals R and S disconnected. Then, Clauses a, b, c and d are deleted and Links 3, 4 and 6 are removed. Without Link 5, Clauses e and f will become pure, and finally be deleted. The final result for the inference on this version of the graph is an empty set which means the original clause set is logically satisfiable. On the contrary, if Link 5 were established, the final result would be an empty clause which implies that the original clause set is unsatisfiable. Such a contradiction is a result of logical inconsistency.

3.2 Example 2

One may speculate that logical inconsistency in the previous example is due to the close proximity of the two links being resolved upon. In this example, we will see that resolutions upon two links that are far apart may also cause logical inconsistency. The input in this example consists of those darkly-circled clauses in Figure 2. Resolutions on Link 3 and Link 7 are performed by two processors concurrently. Resolvents, Clause c and Clause j, have just been generated, but not yet connected. Both Links 3 and 7 were deleted. Now, an attempt to inherit Link 6 for Clause j failed due to an incompatible substitution (i.e., Clause g and Clause j can not be unified because b $\neq$ d). The failure fires a snowball effect, and Clause i, j, h, g, f and b are deleted in sequence. Then, Clause a is checked for pure literals. If the resolvent, Clause c, has not been connected to literal S(x) at this moment, all the clauses will be deleted. The result is an empty set. If link 9 has been established before the snowball effect propagates to Clause a, then the resulting clause set would consists of three clauses, Clause a, Clause c and Clause d. Subsequent resolutions will bring this clause set to an empty clause. Thus, Link 9 is crucial in this example. A logical inconsistency occurs if it can not be established in time.

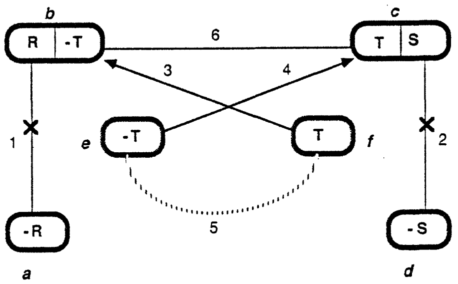


Figure 1. A parallel resolution procedure fails to establish link 5, and results in logical inconsistency

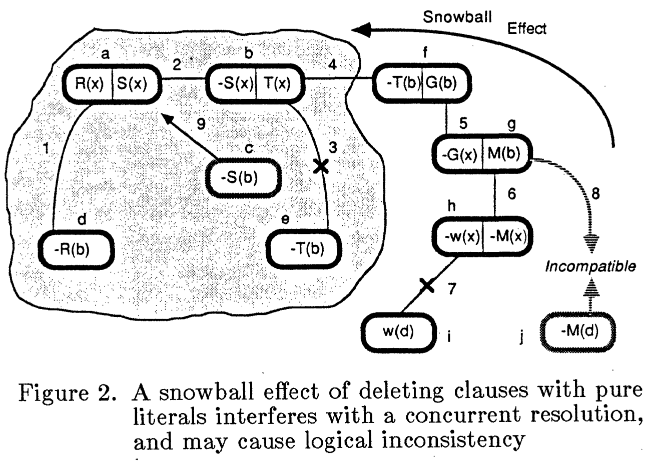


Figure 2. A snowball effect of deleting clauses with pure literals interferes with a concurrent resolution, and may cause logical inconsistency

3.3 Previous work

It has been speculated that the logical inconsistency problem stems from concurrent resolutions on links that are too close to each other. In his dissertation, Loganantharaj proved that two resolutions upon two links can be done concurrently if their respective parent clauses are not mutually connected [11]. He called this condition a *dcdp parallelism* in comparison with AND parallelism and OR parallelism. He then proposed a graph-coloring method to divide the links in the graph into disjoint sets so that multiple resolutions in the same set can be done concurrently without running into the danger of logical inconsistency. There are two drawbacks in this approach. First, overhead is high due to the high complexity of graph coloring. Second, dcdp parallelism is also prone to logical inconsistency when incompatible MGU substitution occurs during link inheritance. Incompatible substitution is common and can start up a snowball effect as shown in the second example above.

4. What Leads to Logical Inconsistency?

4.1 Independence in resolution principle

According to the completeness theorem of resolution principle [4], links should be allowed to resolve upon in an arbitrary order and still obtain a correct result (provided that no particular link is indefinitely excluded from being selected) [16]. The order should affect only the number of resolution cycles incurs in an inference process, which means that some procedures will take longer than others to finish. This property implies that resolutions upon different links are independent. In other words, the resolution principle imposes no precedence constraint on any two resolutions. Since parallel processing also asserts the principle that two independent operations should be capable of being performed concurrently without synchronization. Why does logical inconsistency occur when parallel resolutions are performed on two links in a close proximity? Furthermore, how close can two clauses be without risking the danger of having parallel resolution result in logical inconsistency? It is the objective of this paper to provide some insights into the problem of logical inconsistency.

4.2 Why does logical inconsistency occur?

To investigate why logical inconsistency occurs in a connection-graph-based resolution procedure, we should examine the differences between it and more conventional approaches.

In a conventional resolution procedure, the pair of unifiable clauses to be resolved upon is selected from the clause set using a *search procedure.* Once the pair of clauses is determined, the resolution is performed as an indivisible operation. No other processors will cut in and interfere the resolution. The indivisible resolution is possible primarily due to the way the clause set is organized. The clause set is not structured. It is simply a collection of clauses. A clause can be added to the set or duplicated without affecting the unsatisfiability of the set. Clauses are deleted from the set by a separated *pruning procedure.* Thus, resolutions can be done independent of the clause set once two parent clauses are determined. Although the resolution may be temporarily suspended by interrupts to the processor executing it, its process context will not be changed by other processors. There is no problem for two processors to work on the same clause either since the clause can be duplicated and used as a read-only object.

The story is quite different in a connection-graph-based parallel resolution procedure. Searching for unifiable clauses is eliminated by imposing a connection graph over the clause set. The connection graph is a data structure shared among all the participating processors. Concurrent updates to a shared data structure involve important synchronization issues. Moreover, clause pruning is done in companion with each resolution. The engagement of resolution and clause deletion make graph updates more complicated.

In short, the connection graph approach plays two important roles in: (1) it eliminates the need to search for unifiable clauses, and (2) it deletes redundant clauses. Consequently, a resolution depends heavily on the graph. It includes several steps of graph manipulation, e.g., link inheritance and deletion of pure-literal clauses as we have seen in Section 2. A resolution process becomes vulnerable when its connection graph is arbitrarily changed before resolution has been completed.

5. Parallelism in Concurrent Resolutions

5.1 Bernstein conditions

A set of conditions that are necessary to guard a shared object against malicious access was proposed by Bernstein, and is commonly known as *Bernstein conditions.* It has been shown that data consistency can be maintained as long as Bernstein conditions are satisfied [2, 6].

Since parallelizing a sequential process may change the order of execution, accesses to data may also become out of order. An out-of-order access may result in data inconsistency. For example, assume that a data element is read before it is written in a sequential execution. A corresponding parallel execution would be incorrect if it changed the order of access to be a write and then a read. Thus, if a shared data is involved, certain order must be preserved for a correct parallel execution. This observation leads to the formulation of Bernstein conditions.

Bernstein Conditions:

Let D_i and R_i be the domain and range of an operation, say P_i, respectively. Then, two operations P_i and P_j can be executed concurrently if the following conditions are satisfied:

(1) $D_i \cap R_j = \phi$

"Nothing in the domain of P_i can be in the range of P_j"

(2) $R_i \cap D_j = \phi$

"Nothing in the range of P_i can be in the domain of P_j"

(3) $R_i \cap R_j = \phi$

"Nothing in the range of P_i can be in the range of P_j"

These conditions exclude the possibility of executing two operations concurrently unless no shared data is involved or the data is used without modification. Usually, the terms "domain" and "input" are interchangeable, and so are the "range" and "output".

Bernstein conditions are a set of general constraints valid for all types of parallel processing, and connection-graph-based parallel logic inference is only a special case of parallel processing. Therefore, logical inconsistency due to the false manipulation of links in a connection graph can be avoided if Bernstein conditions are satisfied during all parallel operations.

With the Bernstein conditions, we can re-examine why logical inconsistency occurred in the previous examples. In the first example, two resolutions were carried out by two processors concurrently. When the first processor tried to establish links for its resolvent (i.e., Clause e), it took Clause b and Link 6 as inputs and made Clause c and the new link its outputs. Meanwhile, Clause c and clause b were members of the second processor's input and output sets respectively. Concurrent execution of these two resolutions violates two of the Bernstein conditions, i.e.,

$$R_1 \cap D_2 = \{\text{Clause c}\}$$
$$D_1 \cap R_2 = \{\text{Clause b}\}$$

Note that although Link 6 is in the domains of both resolutions, it does not violate any Bernstein condition. Thus, it is not the link that causes logical inconsistency problem.

In the second example, logical inconsistency is due to the fact that Clause a appears in the range of the link-inheritance operation being performed by processor 1 and in the range of the snowball effect being propagated by another processor.

5.2 Precedence constraints on connection-graph resolution

As described in Section 2, a resolution cycle involves consists of five major steps. The domain and range of each step are summarized in Table 1. Dependence relation of these steps can be established by verifying their input and output against Bernstein conditions. A precedence graph summarizing the dependence relation is shown in Figure 3. Steps that can be done in parallel are represented as different paths in the graph. According to this graph, the parallelism between these resolution steps is limited since there are only two parallel paths. Thus, it is reasonable to treat a resolution cycle as a basic scheduling unit in task assignments. However, we don't include the possible implementation

Step	Operation	Domain (Input)	Range (Output)
1	Generates resolvent	Parents, Link	New clause
2	Tautology check	Resolvent	Resolvent
3	Link inheritance	Parents, Links, Neighbors	Neighbors,Links, Resolvent
4	Link removal	Link, Parents	Link, Parents
5	Clause deletion	Entire graph	Entire graph

Table 1. Domain and range of resolution steps in a resolution cycle

of finer-grained parallelism at lower-level abstraction [1]. For example, the unification procedure in the step of resolvent generation can be parallelized. Also, the parallelism within the step of clause deletion is substantial since multiple clauses can be deleted concurrently. We will further elaborate this point later in this section.

Note that the removal of the resolved link can be done either before or after tautology check and link inheritance. The order of these steps, however, may be crucial under certain circumstances. In Example 2

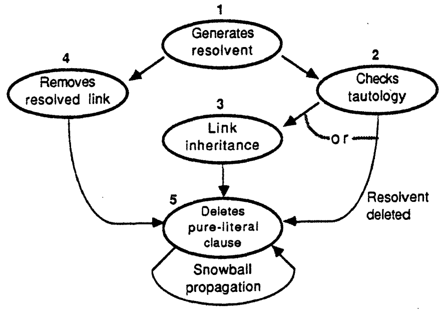


Figure 3. Precedence graph of resolution steps in a resolution cycle

above, for instance, if Link 3 were not removed before link inheritance was done, the propagation of clause deletion would stop at b. Logical inconsistency would not have occurred in that case.

We now examine the dependence relation between resolutions by treating a resolution cycle as a task unit. In the first example in Section 3, the two links resolved upon are at a distance of two hops apart. The two resolutions are designated as L and R respectively, and their steps are labeled by L_i and R_i for i=1,2,...5 for convenience. Because the completeness only guarantees that sequential execution of L before R or R before L is correct, it does not guarantee a correct result if L_i and R_i are interleaved. Therefore we must start from the sequential order of $L_1,...,L_5$, $R_1,...,R_5$ (or $R_1, \dots, R_5$, $L_1, \dots, L_5$) and then try to explore possible parallelism from there based on Bernstein conditions.

The input and output of a resolution step are represented in a form of [inputs/outputs]. An overlap of input/output between steps of L and R is represented by a set {...}. If an overlap is not empty, the two operations can not be executed in parallel since it will violate Bernstein conditions. Thus, a precedence constraint (represented by an arrow) must be imposed. For example, Clause b is in the input of L_1 and in the output of R_3. So, an arrow between L_1 and R_3 is introduced, and it is labeled with {b}. The direction of the arrow is from L to R if L is executed before R in sequential resolution, and from R to L if R is executed before L. Once the order is determined all the other arrows will be in the same direction. Similarly, we can add the following arrows: L_3 to R_1 labeled with {c}, L_3 to R_3 labeled with {b,c}, L_3 to R_5 labeled with {c,4,6}, L_5' to R_3 labeled with {b,3,6}, and L_5' to R_5 labeled with {6}. These arrows represent the set of dependence constraints for a parallel resolution of L and R. A precedence graph summarizing these dependence constraints is given in Figure 4.

According to the resulting precedence graph, R_1 can not be executed before L_3 is finished, and R_3 must be executed after L_5'. These two dependence constraints are so strong that only R_1 and R_2 can be executed in parallel with L. Unconstrained execution of L and R in parallel can run into logical inconsistency problem easily. This explains why two links to be resolved upon concurrently should not be in close proximity.

The second example shown in Figure 2 has a longer distance between the two links being resolved upon. We can also derive a precedence graph for the two resolutions. The result is shown in Figure 5. In this figure, we can see several arrows pointing from L_i' s to R_5.

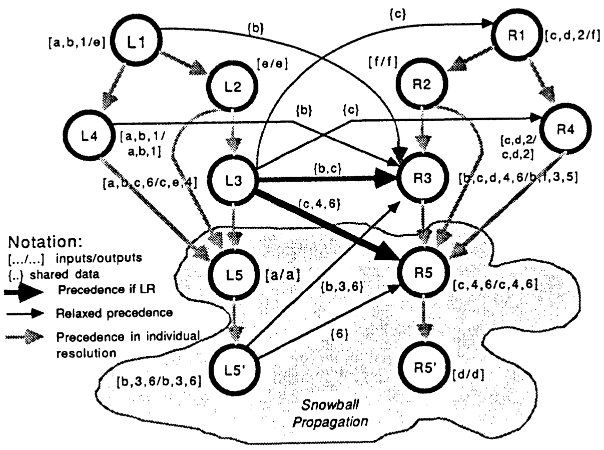


Figure 4. Precedence graph of the two concurrent resolutions in example 1

These precedence constraints protect the resolution L from being interfered with by a snowball effect originating from a remote link. The precedence graph also demonstrates that two concurrent resolutions far apart can also lead to logical inconsistency.

The above precedence graphs were derived by strictly following Bernstein conditions with a simple description of clauses and links (i.e. a,b,... for clauses and 1,2,... for links). As a matter of fact, the dependence constraints can be relaxed. If the data structures representing a clause and its associated links are separately maintained, adding and deleting a link will not modify the clause itself. Thus, a link can be added to or deleted from a clause even when the clause is being used for generating a new clause. The separation of links from clauses makes it possible to relax the dependence constraints $L_1 \rightarrow R_3$, $L_3 \rightarrow R_1$, $L_3 \rightarrow R_4$ and $L_4 \rightarrow R_3$ in Figure 4.

In the next section, we will show that clause deletion blocked by another resolution can be resumed automatically. We will also show that adding an inherited link to a clause will not prohibit the clause from being deleted. Therefore, link inheritance can be done in parallel with clause deletion, and the two arrows originating from L_5' in Figure 4 can be removed. Furthermore, $L_3 \rightarrow R_5$ is implied by $L_3 \rightarrow R_3$ due to transitivity, therefore it can be removed. The only precedence constraints left now is $L_3 \rightarrow R_3$. This says that link inheritance is the only vulnerable step in a resolution cycle that has to be protected.

5.3 Parallelism in snowball propagation

Potential scope of a snowball effect is unlimited. It may affect the entire graph. To ensure a correct precedence, it requires all of the clauses in the graph to be reserved before clause deletion can start. If we did this however, the resulting resolution procedure would become purely sequential if one does. Fortunately, the following theorems show that clause deletion can be done one by one. In the case where the propagation of clause deletion is blocked by another resolution, the deletion process can subsequently it can be resumed by the resolution that stopped it. From a processor point of view, this is a very nice property because it can terminates its deletion procedure whenever the procedure is blocked. The three theorems are summarized and proved below.

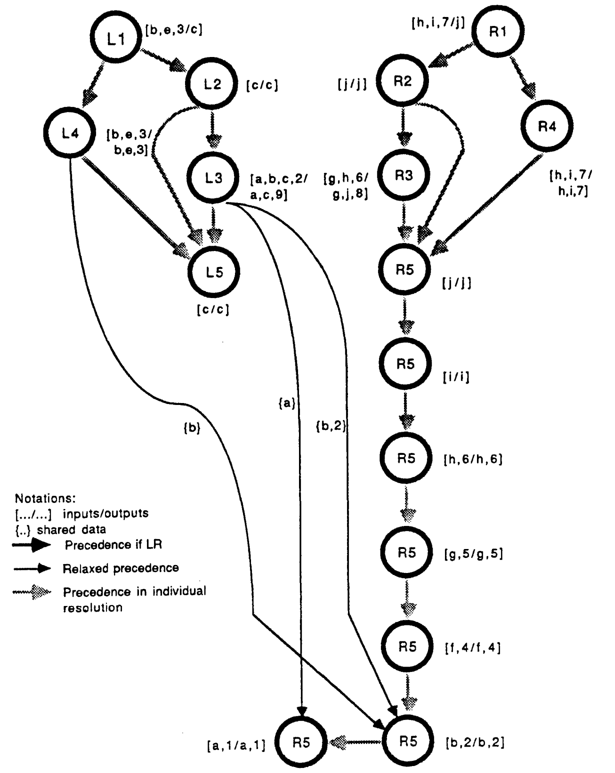


Figure 5. Precedence graph of the two concurrent resolutions in example 2

For convenience, the following terms are defined. Let $c_1, c_2, \ldots, c_k$ be a set of clauses. If c_{i+1} is connected to c_i by a link (c_i, c_{i+1}) such that (c_i, c_{i+1}) is an unique link incident to a literal in c_{i+1}, then such a link is called a *D-link* from c_i to c_{i+1}. Accordingly, a literal becomes pure if the D-link incident to it is removed, and the clause containing it can be deleted. If, for all i $(1 \leq i \leq k)$, c_i and c_{i+1} are connected by a D-link, then $c_1 c_2 \cdots c_k$ is called a *snowball stream.* A snowball stream is a path along which clauses can be deleted successively due to pure literals.

Obviously, if two snowball streams do not intersect, then no precedence constraint exists between clause deletion in each of them. Thus, concurrent clause deletion is possible. If two snowball streams do intersect, then they are dependent. The first theorem below characterizes the concurrent clause deletion in such a case.

Theorem 1:

Clause deletion can be done concurrently along multiple snowball streams even when they merge at some points

<Proof>

Two snowball streams may merge at a clause in two different ways. They can meet either at the same literal (e.g., S_1 and S_2 in Figure 6) or be incident to different literals (e.g., S_1 and S_2 in Figure 6). The former is called an "AND" type of merge, and the latter is called an "OR" type for convenience. In a sequential resolution, clause deletion coming from a single stream of an OR merge will propagate immediately to all down stream paths since the merging clause will be deleted and the first clause of each down stream becomes pure.

For an AND merge, all incident links must be removed before a snowball effect can propagate to down stream paths.
Now consider concurrent clause deletion propagating from multiple streams. Clause deletion from an OR stream will remove an incident D-link and cause a pure literal. The merging clause will be deleted. Deleting the merging clause will fire a snowball effect to all the down stream paths, and remove all the incident D-links. No clauses will be left undeleted since all the rest can propagate up to their D-links incident to the clause. In case an incident D-link of an AND stream is removed first, it may reserved the merging clause, and block an OR stream from propagating snowball effects to down stream paths. However, in spite of being blocked, deletion from an OR stream will remove its D-link and leave a pure literal in the merging clause. The deletion procedure from the AND stream will check for pure literal after it removes the incident D-link. Therefore, the down-stream clauses will always be deleted as they should regardless the type of merge at a given clause. And, snowball propagations upstream do not interfere with each other. Thus, deletion of clauses containing pure literals can be done concurrently.

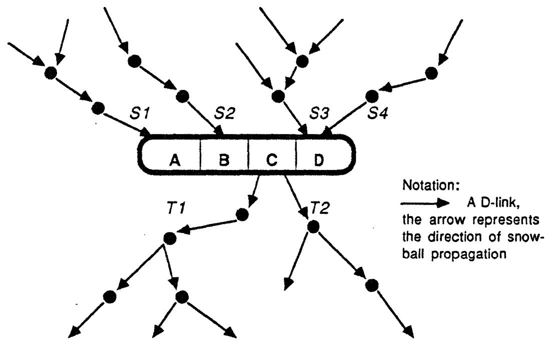

S1
S2
S3
S4
A
B
C
D
T1
T2
Notation:
A D-link, the arrow represents the direction of snowball propagation

Figure 6. Merging and branching of multiple snowball streams

Now consider the dependence between clause deletion and a resolution. It can be characterized by the next theorem.

Theorem 2:

Let $c_1c_2...c_k$ be a snowball stream, and (c_i,c_{i+1}) be a D-link between c_i and c_{i+1}. A resolution performed on the D-link (c_i,c_{i+1}) will cause either (1) the stream to be re-configured to a new stream $c_1c_2 \cdots c_{i-1}\overline{c_i}c_{i+2} \cdots c_k$, where $\overline{c_i}$ is the resolvent, or (2) the stream to be broken into two streams with the first clause in the down stream containing a pure literal.

<Proof>

Consider the situation after the resolvent $\overline{c_i}$ has been generated. The resolvent is subject to tautology check. If tautology is detected, then the resolvent is deleted, otherwise it is eligible to inherit (c_{i-1},c_i) as an incoming D-link and (c_{i+1},c_{i+2}) as an outgoing D-link. There are two possibilities in this case: both links are inherited successfully or at least one of them is incompatible. In case the resolvent is not a tautology and both links are inherited successfully, the snowball stream is reconnected. For the rest, the snowball stream is broken. These

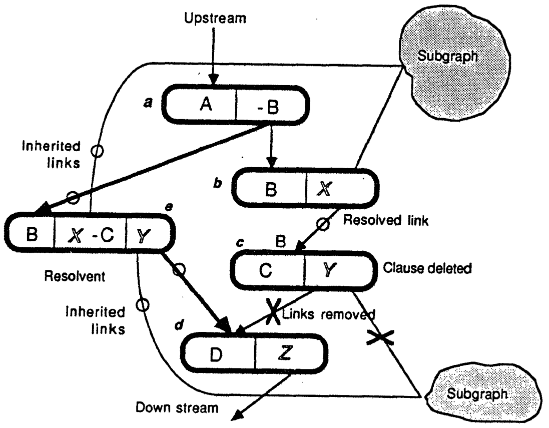


Figure 7. Reconnection of a snowball stream due to a resolution on the path

phenomena are examined respectively below.

(1) The resolvent is not of tautology, and both links are inherited successfully:

The literal in c_{i+2} to which the new outgoing link incident was singularly connected by the D-link (c_{i+1},c_{i+2}) on the original snowball stream. Since the resolved link is a D-link, Clause c_{i+1} will contain a pure literal and be deleted after the removal of the resolved link. Together with it, the D-link (c_{i+1},c_{i+2}) will also be removed. The literal will again become singularly connected by the new link $(\overline{c_i},c_{i+2})$ only. Thus, the new link is a D-link. On the other hand, the incoming link is also a D-link. This is because the resolvent is formed by removing the two complimentary literals connected by the resolved link from the concatenation of the two parents, and the new inherited link is connected to the literal of the same predicate symbol that the original D-link is connected to. Since the resolvent $\overline{c_i}$ is connected to c_{i-1} and c_{i-2} via a D-link respectively, a new snowball stream $c_1,c_2, \cdots c_{i-1},\overline{c_i},c_{i+2}, \cdots c_k$ is established. The original snowball stream is broken after the removal of c_{i+1} and its associated links. Figure 7 illustrates the above reconnection, in which c_{i-1},c_i,c_i, and c_{i+2} are labeled as Clauses a, b, c, and d respectively, and the resolvent $\overline{c_i}$ is shown in shade with a label e.

(2) The resolvent is of tautology or either link is incompatible:

If the resolvent is deleted due to tautology, then no new link is to be added to the graph. Clause c_{i+1} and its associated links are to be deleted since the literal connected by the resolved link becomes pure. Thus the snowball stream is broken into two segments: $c_1,c_2, \cdots c_i$ and $c_{i+2}, \cdots c_k$. The literal in c_{i+2} that was connected by (c_{i+1},c_{i+2}) becomes pure since the link is a D-link and is removed when c_{i+1} is deleted. Now, if the resolvent is not deleted, but the D-link (c_{i-1},c_i) is not inherited successfully, then the snowball stream is also broken. It can be shown using similar arguments to the above that the first half terminates at c_i and the second half starts from the resolvent. If it is the D-link $(c_{i+1}c_{i+2})$ that is not inherited successfully, then the first half terminates at $\overline{c_i}$ and the second one starts from c_{i+2}. Figure 8 illustrates this case. The related clauses are labeled in the same way as in Figure 7. The case in which both links are incompatible follows directly from the two cases above.

The above theorem states that a snowball stream is either reconfigured or broken after a resolution is performed on it. In the latter case, the first clause of the second segment always contains a pure literal. This implies that all the clauses in the down stream path will be deleted due to a snowball effect. The first half will not be affected by the resolution in any case. It may be deleted by some other resolution upstream, or it may remain there if no resolution upstream is done. But, what happens if a propagating snowball effect is blocked by a resolution? Will the rest of the clauses be deleted as they should were the propagation not blocked? The following theorem answers this question.

Theorem 3:

When a propagating snowball effect is blocked by a resolution being performed on a D-link in the middle of a stream, it will be resumed by the blocking resolution.

<Proof>

Again we assume that a resolution is performed on the link (c_i, c_{i+1}). Since only two clauses in the upstream will possibly be reserved during resolution, the coming snowball may be blocked either at c_{i-1} or at c_i depending on which step of the resolution is being executed. If it stops at c_i then c_{i-1} must have been deleted. In this case, the resolution proceeds as though c_{i-1} does not exist and c_i contains a pure literal. The resolvent will also contains a pure literal because it is derived from c_i. Thus, no matter how the snowball stream is broken or reconnected, the down-stream clauses will be deleted when the resolution starts its next step of clause deletion. However, the resolution will not check c_{i-1} for deletion in its original form. If the incoming snowball is blocked at c_{i-1}, it may not be re-started by the current resolution. Fortunately, this can be taken care of by a simple extension which detects pure literals for those clauses reserved for link inheritance.

In summary, the three theorems above indicate that clauses that are supposed to be deleted due to snowball effect will always be deleted wether snowball effect propagation is interfered with by another propagating snowball or disrupted by a resolution. Propagation can be done by reserving clauses one step at a time. No global reservation is needed. The processor executing it can simply leaves the propagation to others without waiting for the reserved clauses to become available.

6. Architecture Support for Parallel Resolution

As *mutual exclusion* is necessary for enforcing precedence constraints, a resolution must enter a *critical section* whenever it intends to operate on the connection graph. To do so, a protocol that requests for the permission must be provided. In this section, several approaches are suggested.

6.1. Lock and Wait or Lock and Withdraw

The *lock and wait* approach is the concept behind many synchronization protocols for critical section management [12]. However, a *lock and wait* scheme may run into deadlock. Consider two concurrent resolutions, each locks one clause and tries to reserve another that happens to be locked by the other resolution. The two resolutions will wait for each other to release locked clauses forever, and thus a deadlock occurs.

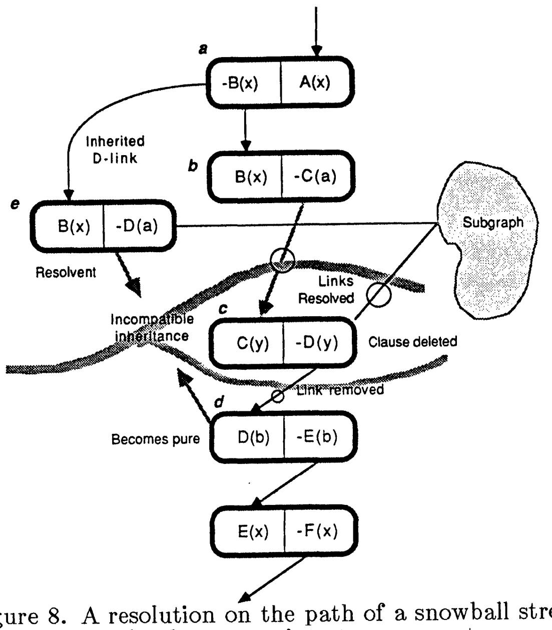


Figure 8. A resolution on the path of a snowball stream breaks the stream into two segments

Deadlock can be avoided by using a *lock and withdraw* scheme. That is, a resolution cycle will be aborted if not all of the required input and outputs are available. Those clauses that are locked will be released if a resolution is aborted. The released clauses can be claimed by resolution being performed by other processors right away. Thus, no deadlock is possible. Nevertheless, after a resolution is aborted, the graph has to be recovered. This may not always be possible. Moreover, a recovery discards all the work that was partially done. Computation power is wasted in this case. For these reason, the feasibility and efficiency of a lock and withdraw scheme has to be examined.

If either of the two parent clauses of a resolution is found to be locked by another resolution on a nearby link, the resolution can be aborted without any problem since nothing has been done yet. The processor can select another link to work on. When a resolvent is newly generated, no link has been established. No locking is necessary in checking tautology. As we can see from Table 1, the first two steps do not modify the connection graph, while the last three steps mainly deal with graph updates. Therefore, when locking fails for step 3, the resolution cycle can also be aborted simply by deleting the unlinked resolvent. Furthermore, we have also shown above that early termination of a snowball propagation will not introduce any new complications either. Thus, it would appear that a lock and withdraw scheme can be effectively applied to the manipulation of connection graphs without any identifiable side effects.

6.2 Graph partitioning

The locking schemes discussed above require that inputs and outputs in every step of resolution are reserved. This requirement presents a nontrivial overhead. The overhead can be substantially reduced by partitioning the connection graph.

The connection graph can be partitioned into as many subgraph as the number of processors available (see Figure 9(b)). Each processor will work primarily on one subgraph. The partition is adjusted dynamically during inference by moving clauses from one subgraph to another as it is needed. For example, if a clause is disconnected from any clause in the same subgraph, then it means it can not be resolved locally. Under such circumstances, the clause has to be sent to another subgraph which has links to it. The migration of clause may also be determined based on the usefulness of the clauses in a subgraph. A migration may be desirable if a clause is more useful in another subgraph. A conceptual clustering scheme that governs graph partitioning and clause migration has been developed and description of the framework can be found in [7]

With dynamic graph partitioning, two resolutions are automatically separated by partition a boundary. There is no danger of overlapping their inputs and outputs except for those clauses near the partition boundaries. Thus, synchronization is needed only when a resolution involves boundary clauses. Since subgraphs can be distributed evenly across different memory banks to diversify memory accesses, this approach may also minimize memory conflicts in a shared-memory MIMD multiprocessor. Memory conflicts is one of the major problems that degrade the performance of such MIMD systems.

6.3 Store and Reconstruction

As a matter of fact, no synchronization is necessary if a proper data structure is established for the links across the partition boundary. The trick is to store all of the changes to a graph locally, and re-construct the part that is to be used. This approach is especially useful in a message-based multiprocessor in which locking across machine boundaries requires sending messages back and forth. Message passing is time-consuming, and should always be minimized.

7. Concluding Remarks

In this paper, we have investigated the parallelism that can be achieved by parallel resolutions on a predicate connection graph. Elimination of searching for unifiable clauses makes a connection-graph-based resolution procedure superior to conventional approaches. With a connection graph, the entire search space is available and is maintained in a well-structured graph. The graph is also kept concise due to the snowball effect of deleting clauses with pure literals. Because of these properties, a connection graph provides a sound basis for parallel logic inference. However, unrestricted parallel resolutions upon the graph may lead to logical inconsistency. This seems to contradict the completeness theorem of logic resolution and the common belief of independence in parallel resolution. Using Bernstein conditions, logical inconsistency is found to be a problem of concurrent manipulations on the connection graph. We have also shown that precedence constraints between two resolutions can be relaxed. Especially, propagation of a snowball effect can be terminated when it is blocked by a resolution. This property allows the proposed *lock-and-withdraw* synchronization scheme to achieve a higher parallelism at its level of abstraction. To further reduce synchronization overhead and memory conflicts, we proposed a graph partitioning approach. For message-based multiprocessors such as the hypercube, we suggest that graph updates across the partition boundaries be stored and re-constructed only when that part of subgraph is to be used. This approach can reduce time-consuming message-based synchronization.

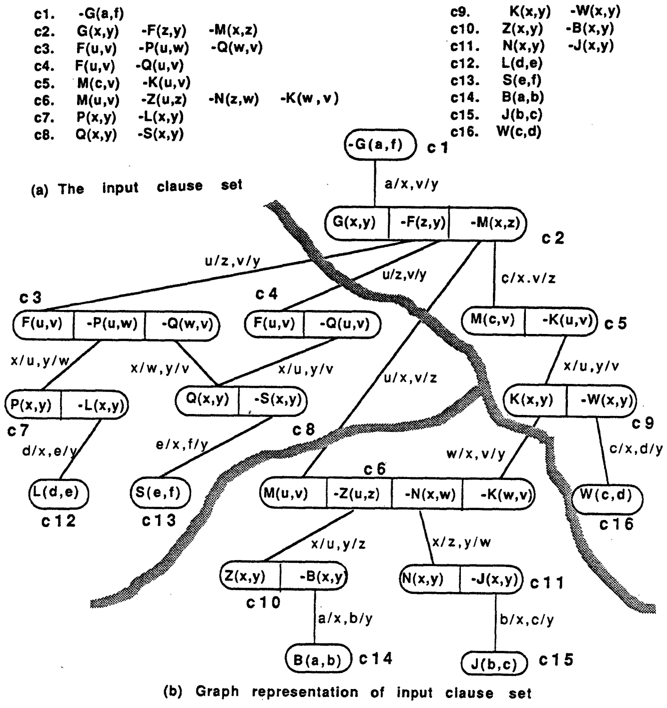


Figure 9. Partitioning a connection graph for parallel resolution

REFERENCES

[1] "A Qualitative Assessment of Parallelism in Expert Systems," *IEEE Software,*, pp. 70-81, May 1985.

[2] A. J. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Tr. Computers*, vol. TSC-15, no. 5, pp. 757-762, Oct. 1966.

[3] W. Bibel, "A Comparative Study of Several Proof Procedures," *Artificial Intelligence*, pp. 269-293, 1982.

[4] M. Genesereth and N. Nillson, in *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Pub. Inc., 1987.

[5] G. Hornung, A. Knapp, and U. Knapp, "A Parallel Connection Graph Proof Procedure," *German Workshop on Artificial Intelligence*, pp. 160-167, Springer-Verlag, Berlin, 1981.

[6] K. Hwang and F. A. Briggs, in *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[7] J. Y. Juang and D. P. Cheng, "A Parallel Inference Model for Logic Programming," *Proc. National Computer Conference* , pp. 87-95, AFIPS, June 1987.

[8] R. Kowaski, "A Proof Procedure Using Connection Graphs," *JACM*, vol. 22, no. 4, pp. 572-595, Oct., 1975.

[9] R. Kowaski, in *Logic for Problem Solving*, 1979.

[10] G. J. Li and B. W. Wah, "MANIP-2: A Multicomputer Architecture for Evaluating Logic Programs," *Proc. Int'l Conf. Parallel Processing,*, pp. 123-130, 1985.

[11] R. Loganantharaj, in *Theoretical and Implementational Aspects of Parallel Link Resolution in Connection Graphs*, Ph.D. Dissertation, Colorado State University, Fort Collins, Colorado, 1985.

[12] J. L. Peterson and A. Silberschatz, in *Operating System Concepts*, Addison-Wesley, 1985.

[13] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *JACM*, vol. 12, no. 1, pp. 23-41, 1965.

[14] S. Sickel, "A Search Technique for Clause Interconnectivity Graphs," *IEEE Tr. Computers*, vol. C-25, no. 8, pp. 823-834, August, 1976.

[15] J. Siekmann and W. Stephan, "Completeness and Consistency of The Connection Graph Proof Procedure," *Interner Bericht*, no. 7/76, Institut fur Informatik, Universitat Karlsruhe, 1976.

[16] J. Siekmann and G. Wrightson, "Paramodulated Connection Graph," *Acta Informatica*, vol. 13, pp. 67-86, 1980.

[17] B. W. Wah, "New Computers for Artificial Intelligence Processing ," *IEEE Computer*, vol. 20 , no. 1, pp. 10-15, January 1987.

PARALLEL IMPLEMENTATION OF PARLOG

Ian Foster

Department of Computing
Imperial College of Science and Technology
180, Queen's Gate, London SW7 2BZ
itf@doc.ic.ac.uk

Abstract — A parallel execution model for the parallel logic language Parlog is described. The model is targeted to non-shared memory multiprocessors. It addresses two important issues in Parlog implementation: the *logical variable*, which provides a restricted form of global address space, and the *control call*, a primitive that permits programs to control and monitor subtasks executing other programs. Distributed unification algorithms are described that permit processes located on different nodes to communicate using shared variables. Alternative distributed unification algorithms permit termination and deadlock detection in distributed tasks at the cost of additional communications. A prototype parallel implementation of Parlog based on this model is being used to investigate its efficiency and refine its design.

Introduction

The parallel logic language Parlog [1] is a simple and elegant process-oriented language. Parlog programs describe systems of light-weight processes which execute concurrently, communicate using shared logical variables and synchronize using dataflow constraints. Parlog programs also have a declarative reading as sentences of predicate logic. This facilitates analysis and understanding of programs. The language has been used in applications as diverse as simulation, theorem proving and process control.

Parallel logic languages such as Parlog have much in common with dataflow and functional languages. It might hence appear that similar techniques can be applied to achieve efficient parallel implementations. However, a parallel implementation of Parlog must deal with two sources of complexity not encountered in functional languages. The first is its logical variable, which introduces some of the problems associated with shared data in a parallel environment. The second is the control call, a primitive that permits a Parlog program to control or detect termination of another program's execution. This introduces a need for distributed control and termination detection mechanisms in an implementation.

Taylor *et al.* [9] propose an elegant solution to the first problem, in the context of the related language FCP. They represent variables that are shared by processes located on two or more nodes by a single occurrence of the variable and multiple *remote references* to it. An attempt to access a variable represented by a remote reference is translated into a communication to the node on which the variable is located. These communications are encoded in *distributed unification algorithms.*

A similar approach can be used to implement shared variables in a parallel implementation of Parlog. However, semantic differences between Parlog and FCP mean that different distributed unification algorithms are required. Distributed unification algorithms for Parlog are described in this paper.

The complexity of the control call discourages a direct implementation of its functionality. It is shown in this paper that its distributed monitoring and control functions can be programmed in Parlog, thus avoiding complexity in an implementation. Minor extensions to the distributed unification algorithms support distributed termination and deadlock detection.

The approach to Parlog implementation described in this paper is targeted to **non-shared-memory multiprocessors.** This class of architecture may be characterized as follows:

- a finite number of nodes, connected by a reliable communication network
- no global storage; instead, each node has local storage
- nodes may communicate by message passing

Problems associated with shared memory communications and unreliable communications are not considered.

Parlog

Language Overview

A Parlog program is a collection of clauses which have the form:

$$H \leftarrow G_1,\ldots,G_m : B_1,\ldots,B_n. \qquad m,n \geq 0$$

where H is the clause's head, ':' is the commit operator and the G's and B's are processes. Each clause can be read declaratively as: "H is true if the G's and B's are true". Clauses with the same name and number of arguments are grouped into procedures.

The G's comprise the guard of a clause. For simplicity, this paper deals with the subset of Parlog in which the G's are restricted to be predefined test operations. As full Parlog can be compiled to this subset [6], this does not imply a lack of generality.

The following program illustrates the language.

```
go ← producer(Xs, sync), consumer(Xs).                      (C1)

producer(Xs,sync) ← Xs = [X |Xs1], producer(Xs1).           (C2)

consumer([X |Xs]) ← X = sync, consumer(Xs).                 (C3)
```

The notation [Head |Tail] denotes a list structure with a Head and Tail. Strings beginning with uppercase letters denote variables while those with lower case denote constants.

A call to this program:

?- go.

creates two processes, producer and consumer (C1). producer communicates a stream of messages (unique variables: X, X1, ...) to consumer by incrementally constructing a list structure containing these variables (C2). consumer acknowledges each such 'communication' by assigning each variable it receives the value sync (C3).

Non-variable terms in Parlog clause heads define dataflow constraints: a clause cannot be used to reduce a process until a process' arguments match its own. In consequence, producer and consumer communicate synchronously. consumer waits for a communication from producer; producer then waits for the variable it has communicated to be bound to sync.

Operational Model

The state of a Parlog computation may be represented as a **process pool**. Computation proceeds by repeatedly selecting a process and attempting to reduce it using the clauses in the associated procedure. A reduction attempt may succeed, fail or suspend.

A **process reduction** comprises two phases: *test* and *spawn*.

Test phase: In the test phase, an attempt is made to find a clause capable of reducing the process. Non-variable arguments in the heads of all clauses are *matched* with corresponding process arguments and guard tests are evaluated. Matching and guard evaluation cannot bind variables in process arguments and suspend if applied to variable process arguments. A clause is a **candidate** if both input matching and guard evaluation succeed. If no clause is a candidate and at least one clause has suspended, the reduction attempt **suspends** and the process is put back in the process pool. If no clause is a candidate and no clause suspends, the reduction attempt has **failed**. If one or more candidate clauses are found, the reduction attempt has succeeded. A candidate clause is selected and reduction proceeds to the spawn phase.

Spawn phase: In the spawn phase, unification operations (that is, processes with the form: X = Y) in the body of the selected clause are performed and other body goals are added to the process pool.

Unification is a recursive matching procedure that attempts to make two terms identical, binding variables in either term if necessary. In Parlog, it is generally used to simply assign a value to a variable.

Consider the producer procedure above (which consists of a single clause). This can be used to reduce a producer process if the test:

Arg2 = sync

succeeds, where Arg2 represents the second argument of the process being reduced. If Arg2 is not instantiated, this test and hence the producer process suspend.

Assume that producer's second argument has been assigned the value sync. The test succeeds, allowing the process to be reduced: the unification operation Xs = [X |Xs1] is performed and the process producer(Xs1,X) is added to the process pool.

Implementations of Parlog generally optimize this simple operational model by introducing a **suspension structure** which permits suspended processes to be associated with variables for which they require values. Such processes are not selected for reduction again until one of these values becomes available. This avoids the overhead of repeatedly selecting a suspended process for reduction.

A Parlog computation terminates when:

- the process pool is empty, in which case the computation has **succeeded.**
- a reduction attempt or unification operation fails, in which case the computation has **failed.**
- there are no reducible processes, yet the process pool is non-empty, in which case the computation has **deadlocked.** (Because of dataflow constraints, every process is suspended waiting for data).

Controlling Computation

An important component of the Parlog language is its **control call.** A call to this primitive has the general form:

call(Module, Process, Status, Control)

It denotes the controlled execution of Process using the program defined in Module. Another, concurrent process can subsequently suspend, continue and abort evaluation of Process by binding Control to a stream of messages. It can monitor its progress by inspecting Status. This variable is bound to a stream of messages to report both termination and run-time errors.

The control call provides Parlog with the important notion of **task:** a unit of computation that may be monitored and controlled as a single entity. The operational effect of the control call is to create a subtask or process subpool within the process pool in which it is called.

Distributed Unification

Consider the problem of achieving a *multiprocessor* implementation of Parlog in which the process pool representing a task may be distributed over several nodes in a multiprocessor. Assume that a *uniprocessor* implementation of Parlog exists on each node [3,5]. This supports process reduction, unification and the control call. To support parallel execution, this must be extended to provide:

distributed unification: to allow processes located on different nodes to communicate using shared variables.

distributed control: to control and detect termination and runtime errors in tasks distributed over several nodes.

One problem that does *not* need to be dealt with in an implementation is load balancing. This is programmed in the language using message passing.

Recall that Parlog's computational model distinguishes between test and unification operation on variables. Assume that, as in Taylor *et al.*'s FCP implementation, variables shared by processes located on several nodes are represented by a single occurrence and several remote references to this occurrence. In a multiprocessor, both *tests* and *unifications* can encounter remote references. When this occurs, distributed unification algorithms are invoked in the implementation.

The *unify* algorithm is applied during the *spawn* phase of Parlog reduction when unification operations encounter remote references. It generates messages to the nodes on which remote terms are located requesting that they perform unification operations.

The *read* algorithm is applied during the *test* phase when tests encounter remote references. A *test* operation applied to a remote reference is made to suspend as if the remote reference denoted a variable. If the reduction attempt suspends, messages are generated to request the values of any remote terms encountered in suspending clauses. The process is associated with these remote references in the suspension structure; it will thus not be selected for reduction until a requested remote value is returned. (The term 'value' is defined below).

It is useful to distinguish between strict and non-strict tests. Most Parlog tests are **strict**: they suspend until all input arguments are instantiated and then succeed or fail. For example, integer or atomic. A few, such as Parlog's equality and inequality primitives == and =/=, are **non-strict**: they can *sometimes* proceed when their input arguments are variables. For example, a call X==Y is defined to succeed if its two arguments are syntactically identical; this includes the case when they are identical variables. A call X==Y can thus succeed when its arguments are both variable, if they are the *same* variable.

Requests generated following strict and non-strict tests are distinguished. As a strict test cannot proceed until a variable argument is bound, it is not necessary to return the value of a remote term required by a strict test until it is non-variable. A non-strict test, on the other hand, needs to know if the remote term is a variable and, if so, what is its location. (For example, a call X==Y must succeed if X and Y are remote references to the same variable). The value of a remote term must thus be returned immediately when required by a non-strict test, even if it is variable. To implement this distinction, the *ns_read* algorithm is defined, to be applied instead of the *read* algorithm when a non-strict test encounters a remote reference.

Following sections define the *read*, *ns_read* and *unify* algorithms in terms of:

- *when* messages are generated, and
- *what* messages are generated, and
- *how* messages are processed.

More detailed descriptions of the algorithms can be found in [4].

Each node in a multiprocessor is assumed to alternately perform reduction attempts and process incoming messages. Processing a message may generate further messages and/or modify internal data structures. Alternation of reduction and message processing ensures that internal data structures are not left in inconsistent states.

Messages are represented here as structured terms: read(T,F), unify(X,T,Y), etc. The functor represents the message type; the first subterm (T, X, etc.) is always a remote reference (that is, a {node, location} pair) representing the *destination* of the message.

Messages are generated as a result of test and unification operations or whilst processing messages. When a message is generated, it is sent to the node referenced by its first component. A node receiving a message examines the location referenced by this component. If this is a remote reference, the message is forwarded: this dereferences chains of remote references. Otherwise, the node can proceed to process the message. Dereferencing of remote reference chains is assumed in the following descriptions and is not mentioned explicitly.

A chain of remote references may lead a message back to its source node, making an apparently remote operation a local operation [9]. For clarity of presentation, the descriptions that follow ignore such special cases. Only minor modifications to the algorithms are required to deal with them.

Strict Tests: The *read* Algorithm

If a *strict* test requires the value of a remote term, a read message is generated to the node indicated in the remote reference. A node receiving a read message returns the value of a *non-variable* term immediately using a value message. The value of a *variable* is not returned until the variable is bound. A **broadcast note** is attached to the variable to record the pending request. It is thus necessary to check for broadcast notes when binding variables. Pending requests are responded to with value messages.

The value of a term is defined to be the scalar value of a constant, one or more levels of a structure (including constant subterms and remote references to other subterms) and a remote reference to a variable.

A value message copies the value of a term from one node to another. A node receiving a value message replaces the remote reference with the value and awakens any processes suspended waiting for it. Subsequent accesses to a non-variable value do not require communication. This copying of non-variable terms is possible because of the single-assignment property of Parlog variables.

Note that a read message is only generated for the first process to suspend on a particular remote reference on a particular node.

The read and value messages have the form:

```
read(To, From)
value(From, Value)
```

where To is a remote reference to the remote term (that is, a representation of its node and location), From represents the node and location of the original remote reference and Value is the value of the remote term.

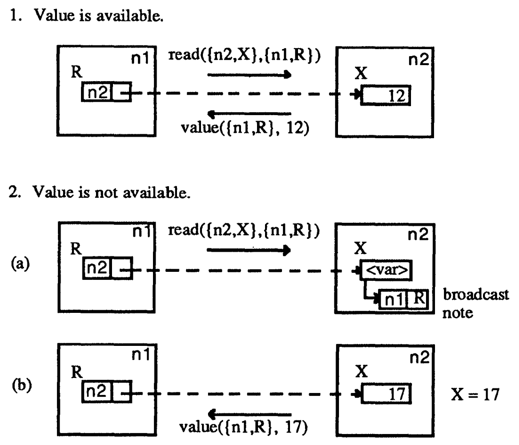


Figure 1 The *read* distributed unification algorithm.

Figure 1 shows two examples of remote reading. In each case, the value of a term represented by a remote reference (represented here as {n2, X}: that is, location X on node n2) is required by a strict test (for example, integer(X)). The value of X is requested using a read message. In the first example, the remote term is available (it is the integer 12) and is returned immediately using a value message. In the second example, the remote term is not available: location X is a variable. A broadcast node is therefore associated with the variable (a). When this variable is instantiated (X=17), its value is returned using a value message (b).

Non-Strict Tests: The *ns_read* Algorithm

If a *non-strict* test requires the value of a remote term, a ns_read message is generated to the node indicated in the remote reference. If the remote term is *non-variable*, its value is returned using a value message, as before. If it is a *variable*, a broadcast note is attached to the variable *and* a remote reference to the variable is returned in a ns_value message. The node that initiated the request can then replace the initial remote reference (which may have been the head of a reference chain) with this direct remote reference. Non-strict tests that required the remote term may then be repeated. If they still suspend, there is no need to request the value of the remote term again. The broadcast note attached to the remote variable means that its value will be returned as soon as the variable is bound.

A ns_read message is generated for the first process to suspend on a particular remote reference because of a non-strict test.

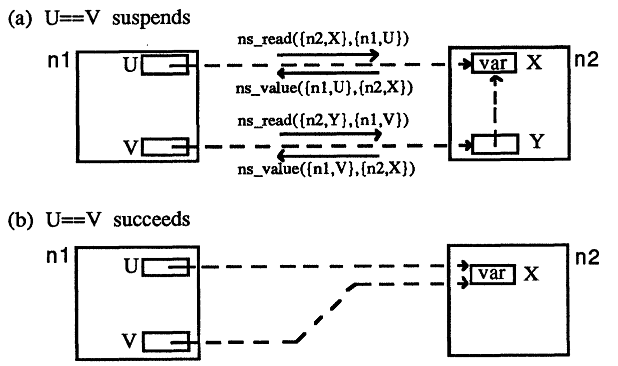


Figure 2 The *ns_read* distributed unification algorithm.

Figure 2 illustrates the use of ns_read and ns_value messages. In (a), a non-strict test U==V encounters two remote references. Although these indicate different locations on node n2 they in fact refer to the same variable, X. The test U==V initially suspends and ns_read messages are generated. Reference chains are dereferenced and 'direct' remote references returned to node n1. These replace the original remote references. In (b), the test U==V is repeated and, as U and V are now identical remote references, succeeds.

The *unify* Algorithm

Recall that unification operations have the form X=Y and are performed during the *spawn* phase of Parlog reduction. If a unification operation encounters a remote reference, a unify message is generated to request nodes on which remote term(s) are located to continue unification. Failure of such a remote unification operation is signalled to the node on which it was initiated by a failure message. This permits an error message to be signalled on the status stream of the task in which the unification operation was performed.

Consider a unification operation X=Y. If *one* of the terms X or Y is represented by a remote reference (say X), a unify message is generated to the node referenced by the remote reference. This message carries the value of the other term (Y) to be unified to the node on which the remote term (X) is located.

If *both* terms to be unified are remote, a unify1 message containing remote references to both terms is dispatched to the node on which the first is located; a node receiving such a message forwards a unify message containing the value of that term to the node on which the second term is located.

In both cases, a unify message eventually arrives at a node on which one of the terms to be unified is located, carrying the value of the other term. The unification operation can then proceed.

A unify message has the form:

unify(X, T, Y)

where X is a remote reference to a term, T denotes the task which initiated the unification operation and Y is the value of a term. A unify1 message has the same form: unify1(X, T, Y), but both X and Y are remote references to terms.

A failure message has the form:

failure(T, X, Y)

where T specifies the task which initiated the unification operation (X=Y) that resulted in failure, and X and Y are the values of the terms that could not be unified.

Figure 3 illustrates the messages that may be generated by the *unify* algorithm if one or both arguments in a unification operation are remote terms.

In (a), only one of the terms to be unified, X, is represented by a remote reference. A unify message is generated to node n2. This contains remote references to X and to the task in which the unification operation occurs (T), plus the value of the other term, Y (<yval>).

In (b), both terms to be unified are represented by remote references. A unify1 message is generated to n3, the node on which one of these terms, Y, is located. This carries remote references to X and Y. Node n3 receives the unify1 message, determines the value of Y (<yval>) and forwards a unify message to node n2. As in (a), this contains remote references to X and to the task T, plus the value of the other term, Y (<yval>).

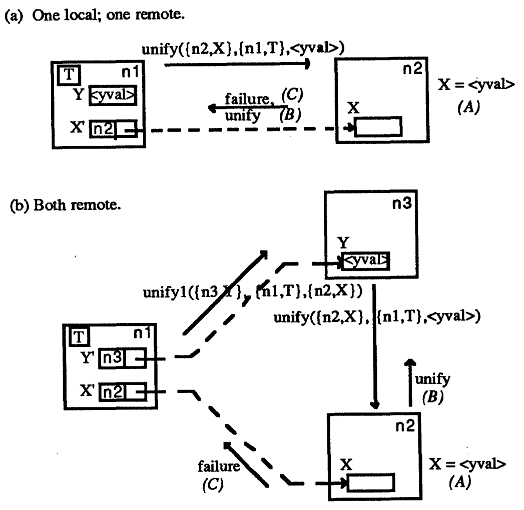


Figure 3 The *unify* distributed unification algorithm.

In both cases, the node n2 on which the term X is located receives a unify message containing the value of Y and performs the actual unification operation. This may generate further unify or failure messages. The unification algorithm applied is described in detail in [4].

Briefly (letters A, B, C refer to Figure 3):

- If X and Y are the same constant, nothing is done.
- A local variable X is bound to reference to a remote constant or structure Y (A)
- If X is a variable and Y is not, a unify message is generated to request the node containing Y to unify Y with X (B).
- If both X and Y are variable, an order check (defined below) is performed.
- If X and Y are tuples of the same arity, corresponding subterms are recursively unified. (Further messages may be generated; these are not illustrated).
- In all other cases, a failure message is generated (C).

Circular References

In logic programming systems, circular references can be created if a variable X can be bound to a variable Y at the same time as Y is bound to X, as illustrated in Figure 4. This problem can be avoided on shared-memory multiprocessors by using pointer comparison to ensure that variable to variable bindings are only created in a certain direction (from low address to high address, for example). A similar technique can be used on non-shared memory machines. An ordering is defined on node identifiers. An **order check** compares node identifiers when variables located on different nodes are unified. Bindings are only permitted from a node of lower identifier to a node of higher identifier.

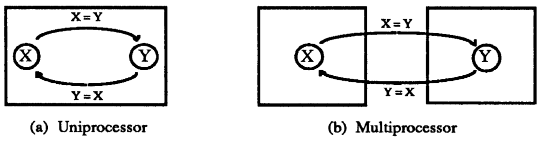


Figure 4 Circular references.

The order check is applied when a local variable X is unified with a variable represented by a remote reference Y. If the ordering constraint is violated (that is, node(X) > node(Y)), the unify message is forwarded to the other node. This causes the unification operation to be repeated in the opposite direction. Otherwise, X is bound to a remote reference to Y (unless both variables are located on the same node). In both cases, the binding is created in the correct direction, from low to high node.

Complexity of Distributed Unification

The program presented at the beginning of this paper is used to illustrate the use of distributed unification algorithms and to motivate some observations on their complexity.

Recall that this program implements synchronous communication between two processes, producer and consumer. Assume that producer and consumer are located on different nodes. Figure 5 illustrates the messages generated when the conjunction is executed by a Parlog implementation using the *unify* and *read* distributed unification algorithms. In (a), the initial situation is represented. It is assumed that the variable Xs is located on the same node as producer; consumer thus possesses a remote reference to this variable. In (b), consumer attempts to read the shared variable Xs, thus causing a read message to be generated to retrieve that value. Meanwhile, producer has generated a value for Xs (say [X |Xs1]). In (c), a value message returns this list structure to consumer; this contains remote references to X and Xs1. In (d), consumer unifies the newly received term X with the constant synch. As X is a remote reference, a unify message is generated to producer.

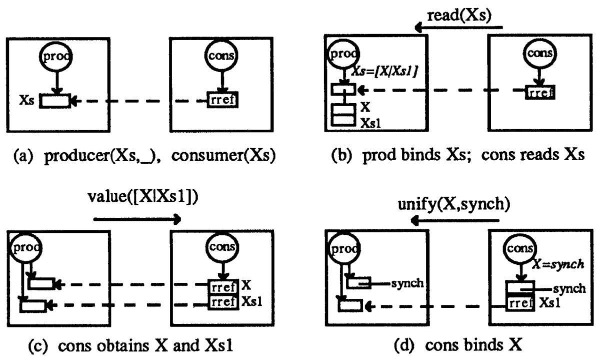


Figure 5 Distributed unification.

The Parlog implementation uses *three* messages to send and acknowledge a 'communication'. Two messages are required to 'read' the original value; one message is required to 'write' the value synch to the variable X. In contrast, if this algorithm were to be implemented in a language with explicit *send* and *receive* primitives, *two* messages would be required for each 'communication': one to send it and one to acknowledge it.

Two points can be made:

- The distributed unification algorithms presented here are *in general* optimal in their communication complexity. That is, $O(N)$ messages are required to communicate $O(N)$ values between nodes. ('In general', because when unifying two variables, order checks may cause additional communications. This is however a special case, as variables rather than values are involved).

- The distributed unification algorithms presented here are *lazy*: a value must be requested by a reader before it is transmitted. This is why three messages are required to communicate two values. There is scope for optimizations that eagerly propagate values when readers are known to require them.

This example also illustrates a useful optimization that can be made in an implementation of the *unify* algorithm. If a remote reference is encountered during a unification operation, a unify message is generated, as in Figure 5 (d). This requests that a remote node unify the remote value (say A) with another term (B). The local remote reference, which represents the remote term A, can *immediately* be replaced with the term with which the remote term is to be unified. (In Figure 5 (d), the string synch). Subsequent references to that term need not therefore generate communications.

Distributed Control

Recall that Parlog's control call provides the following functionality:

- *monitoring*: termination and error detection in tasks
- *control*: the ability to suspend, resume and abort tasks

It is relatively easy to implement these functions efficiently in a uniprocessor [4]. For example, to detect termination of uniprocessor tasks, a **process count** is associated with each task. This is incremented when processes are created and decremented when processes terminate. A process count of zero represents task termination. Providing the same level of control in a multiprocessor is problematic. Existing algorithms for distributed control and termination detection are complex [2]. Incorporating such algorithms in the language implementation compromises simplicity and flexibility. Fortunately, it is possible to program distributed control functions in Parlog.

Assume that each node in a multiprocessor supports a uniprocessor implementation of Parlog, including a control call capable of monitoring and controlling a *uniprocessor* task: that is, a task executing on a single node. Assume also that each node supports distributed unification algorithms that permit processes located on different nodes to communicate using shared variables.

Now consider the problem of monitoring and controlling a *distributed task*: a task executing on several nodes. Observe that the process pool representing a distributed task can be viewed as a number of process subpools, one per node on which the task is executing. For example, in Figure 6, a distributed task T executing on three nodes N1, N2 and N3 comprises three subpools T1, T2 and T3.

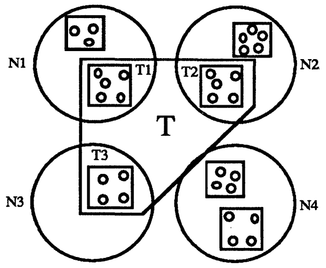


Figure 6 A distributed task.

Assume that processes in these subpools can communicate with processes in other subpools with which they share variables, but cannot migrate to other processes. (This is a consequence of programming load balancing in the language). Then a uniprocessor control call is sufficient to terminate (and detect termination of) all processes in a distributed task. Each constituent subpool is executed as a separate uniprocessor task. Parlog processes are provided that coordinate the monitoring and control of the subtasks. A possible configuration for these processes is illustrated in Figure 7. Supervisor processes (sv) are linked in a circuit using shared variables. A coordinator (coord) provides status and control streams which can be used to monitor and control the distributed task. For example, a request to *terminate* the distributed task (received on Control) is translated by the coordinator into a message which passes around the circuit. A supervisor receiving such a message terminates its subtask using the local control call's control stream (Ci), and forwards the message. When the control message eventually returns to the coordinator, it is known that all subtasks have been terminated.

Termination detection exploits Takeuchi's short circuit technique [8]. If a supervisor detects termination of its subtask, it unifies the two variables forming its part of the circuit. Eventually, when all subtasks have terminated, the coordinator will have two references to the same variable; a test L==R will hence succeed. (Recall that a call to == is defined to succeed if its two arguments are the same variable). Termination can then be signalled on the status stream (Status).

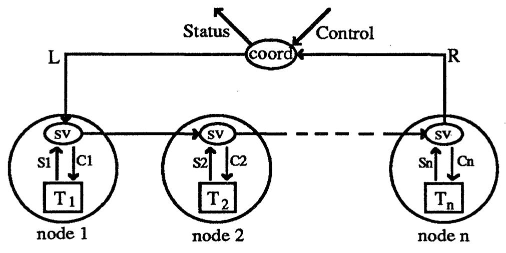


Figure 7 Distributed control.

Both termination and termination detection rely on the fact that once a pool of Parlog processes is empty, subsequent communication cannot create new processes. This is a consequence of programming load balancing in the language. Messages generated by distributed unification algorithms may still be in transit; however, these messages convey data, not processes. Process mapping occurs if this data is *interpreted* as processes by a Parlog process. If no processes remain to interpret the data, no process mapping can occur.

Termination and Deadlock Detection

The uniprocessor control call can be used to program mechanisms that detect when all processes in a distributed task have terminated. Messages generated as a result of unification operations performed by processes in the task may however remain in transit after all processes have terminated.

Recall that four types of message are used by the distributed unification algorithms: read, value, unify and failure. read and value messages cannot affect the course of subsequent computation if the processes that requested the values they are retrieving have terminated. In contrast, unify and failure messages can bind variables or signal errors. A task has not therefore be said to have truly terminated until all its constituent subtasks have terminated *and* all unify and failure messages that it has generated have been processed.

A possible solution to this problem of messages in transit is to implement a global termination detection algorithm that verifies that all unify and failure messages generated by a task are received. A much simpler, albeit somewhat more expensive, solution is to cause each unify message to be acknowledged. This permits the use of message counts associated with individual uniprocessor tasks to detect when all outstanding unify and failure messages have been processed. Termination detection can still be programmed in the language.

The latter solution is used in the execution model reported herein. It is embodied in an alternative *unify* algorithm, *s_unify*, which acknowledges successful remote unifications. This differs from the *unify* algorithm described previously in three respects:

- It uses s_unify and s_unify1 messages rather than unify and unify1 messages.

- It acknowledges successful remote unifications. A node receiving a s_unify message processes it as it would a unify message, but acknowledges successful completion of the remote unification operation using an ack message.
- It does not perform remote unification operations when they involve two structures. Instead, it generates a structure message to return both structures to the node which initiated the unification. This can then initiate new unification operations, one per structure element. This avoids a need for the complex mechanisms that would be required to detect termination of the recursive unification of two remote structures.

Assume that the Parlog implementation on each node in a multiprocessor associates a process count with each uniprocessor task. A task's process count is incremented when it creates a process *or* generates a s_unify message; it is decremented when a process terminates *or* the task receives a failure or ack message. A process count of zero then signifies that both all a task's processes *and* all remote unifications that it has initiated have terminated.

If all subtasks comprising a distributed task use the *s_unify* algorithm, it is known that when all subtasks have terminated all processes in the distributed task have terminated *and* all unify and failure messages generated by this task have been processed. This is true termination. A task for which termination detection is required hence uses the *s_unify* algorithm; other tasks can use the more efficient *unify* algorithm. The algorithm to be used is specified when a task is initiated.

s_unify messages have the same form as unify messages. The ack and structure messages have the form:

```
ack(T)
structure(T, X, Y)
```

where T specifies the location of the task which initiated the remote unification operation and X and Y are the two structures that are to be unified.

The structure message is, strictly speaking, an unnecessary communication. It may thus appear to be a source of inefficiency. However, unification of two structures, though possible in Parlog, occurs rarely in practice. To determine the approximate frequency of such operations, ten Parlog applications (including compilers, programming environments, simulation programs and process control programs, written by different programmers) were tested on an instrumented uniprocessor Parlog implementation. Less than one per cent of all unification operations involved two structures. As remote structure-to-structure unifications are a subset of all structure-to-structure unifications on a multiprocessor, it can be expected that such operations will be extremely rare. The structure message thus appears to be a useful and inexpensive simplification.

Deadlock

Recall that a Parlog computation is deadlocked when all its processes are suspended due to dataflow constraints. Deadlock can easily be detected in a uniprocessor: an active process count is maintained for each task; if this reaches zero and the task has not terminated, the task is known to be deadlocked. Deadlock detection in a multiprocessor is more complicated, again because of messages in transit: it is not immediately possible to determine whether a process is suspended because there is no producer for a remote value or merely because a read or value message is still in transit. In [4], it is shown that an alternative distributed unification algorithm *d_read* can permit a deadlock detection algorithm due to Dijkstra *et al.* [2] to be programmed in Parlog without a need for global message counts. The *d_read* algorithm, which acknowledges certain read and value messages, is used in place of *read* when deadlock detection is required.

Related Work

FCP

Taylor *et al.*'s work on parallel implementation of FCP [9] provided a number of ideas which were exploited in the work reported herein. Key are the idea of using remote references to implement a global address space and the use of broadcast notes to record requests for values that are not yet available. However, the execution model described herein differs from that of Taylor *et al.* in a number of important respects.

One difference is that the algorithms presented here (and others described in [4]) support termination and deadlock detection in distributed tasks. Taylor *et al.* do not address these problems in their paper.

Other differences derive from semantic differences between Parlog and FCP. FCP uses general unification rather than input matching to determine whether a clause can reduce a process. This means that a clause may be required to successfully perform two or more unification operations before it can be selected to reduce a process. These must be performed *as an atomic action*: if any one fails, the others must not occur. To provide this atomicity, the FCP implementation supports *variable migration.* All variables that are to be bound by a reduction are fetched locally before the reduction is performed. To prevent livelock when several nodes require the same variables, variables fetched in this way are locked; this in turn requires a deadlock prevention scheme. Starvation is possible. Once all variables required by a process have been fetched locally, reduction can occur without further communication. Any binding performed during a reduction attempt is recorded and undone if reduction fails. As reduction attempts and message processing are alternated, these 'unsuccessful bindings' are not visible to other processes.

In Parlog, on the other hand, variables are only tested prior to reduction. Test operations are encoded efficiently using read and ns_read messages. Unification is performed after reduction in a number of independent operations. Unification operations involving remote terms are requested using unify messages and variables do not migrate. Alternating message processing and process reduction at nodes provides mutual exclusion when binding individual variables. As processes do not compete for resources (variables), deadlock, livelock and starvation cannot occur.

Experimental studies are required to quantify the run-time costs associated with the FCP and Parlog distributed unification algorithms; these have not been performed. Intuitively, it would seem that the Parlog algorithms are less expensive, as variables do not need to be migrated before being bound, and variable locking and deadlock detection are not required.

FGHC

Ichiyoshi *et al.* [7] describe a parallel implementation of the parallel logic language FGHC. They incorporate distributed control functions such as termination detection in the lowest level of their implementation, rather than programming this functionality in the high-level language as proposed here. Their approach provides efficient support for certain functions but results in a more complex and less flexible implementation.

Conclusions

A parallel implementation of Parlog must support the language's *logical variable* and *control call.* This paper has described simple and efficient treatments of these two language features.

Distributed unification algorithms have been described that permit processes located on different nodes in a multiprocessor to communicate by unifying shared variables. The distributed unification algorithms are simple. There are three basic message types — read, value and unify — plus three acknowledgement messages: failure, ack and structure. The algorithms are in general optimal in their communication complexity. That is, except when order checks fail when unifying variables, there is no 'hidden communication': reading or writing a remote value involves a small, constant number of messages. This means that although Parlog is a high-level language, programmers can visualize the communications implied by programs and can hence implement particular communication algorithms.

The complexities inherent in the distributed monitoring and control functions represented by the control call are not incorporated in the execution model. Instead, these functions are programmed in the language using simple uniprocessor mechanisms to control components of a task located on a single node. Alternative distributed unification algorithm support termination and deadlock detection in distributed tasks. These provide added functionality at the cost of additional communication.

This approach to control call implementation has three advantages compared to a full implementation of the same functions. First, the basic implementation is kept extremely simple. Second, it permits a programmer to trade off functionality and efficiency by an appropriate choice of distributed unification algorithm. Third, it provides greater flexibility. A potential disadvantage of the approach is that these functions may be less efficient when programmed in Parlog than when supported directly in the language implementation. Experimental studies will be performed to determine whether additional costs associated with the approach described herein are significant.

The distributed unification algorithms described in this paper are incorporated in a distributed implementation of Parlog on a network of SUN workstations. It is planned to port this implementation to a non-shared memory multiprocessor in the near future. In the meantime, the SUN implementation is being used for experimental studies aimed at determining the relative costs of the various distributed unification algorithms (and hence the costs of termination and deadlock detection). Another area of ongoing research is simplification of the basic distributed unification algorithms. Minor changes to Parlog's semantics can permit significant simplifications. For example, the *ns_read* algorithm is required to support Parlog's 'non-strict' test operations. If these are redefined to be strict, this algorithm is no longer required.

Acknowledgements

This research was supported by the Science and Engineering Research Council under grant GR/D/24036.

The author is grateful to Steve Taylor for enlightening discussions on parallel implementation. Thanks also to Nobuyuki Ichiyoshi and others at the ICOT research centre for explaining the details of their parallel FGHC implementation and for commenting on an earlier draft of this paper.

References

[1] Clark, K.L. and Gregory, S. "Parlog: parallel programming in logic," *ACM Trans. on Programming Languages and Systems* 8(1) (January, 1986), 1-49.

[2] Dijkstra, E.W., Feijen, W.H.J. and van Gasteren, A.J.M.. "Derivation of a termination detection algorithm for distributed computations," *Information Processing Letters* 16 (1983), 217-219.

[3] Foster, I.T., Gregory, S., Ringwood, G.A. and Satoh, K. "A sequential implementation of Parlog", *Proc. 3rd Intl Logic Programming Conf.*, LNCS-225, Springer-Verlag, (Juner, 1986) 149-156.

[4] Foster, I.T. *Parlog as a Systems Programming Language*, PhD thesis, Imperial College, London (March, 1988).

[5] Foster, I.T. and Taylor, S. "Flat Parlog: a basis for comparison", *International Journal of Parallel Programming*, 16(2) (1988).

[6] Gregory, S. *Parallel Logic Programming in Parlog.* Addison-Wesley, Reading, Mass. (1987).

[7] Ichiyoshi, N., Miyazaki, T. and Taki, K. "A distributed implementation of Flat GHC on the Multi-PSI," *Logic Programming: Proc. 4th Intl Conf.*, MIT Press (May, 1987), 257-275.

[8] Takeuchi, A. 1983. "How to solve it in Concurrent Prolog". Unpublished note, ICOT, Tokyo.

[9] Taylor, S., Safra, S. and Shapiro, E. "A parallel implementation of Flat Concurrent Prolog," *International Journal of Parallel Programming*, 15(3) (1987), 245-275.

Memory Performance of AND-parallel Prolog on Shared-Memory Architectures

M. Hermenegildo – MCC
E. Tick – Stanford University

Abstract

The goal of the RAP-WAM AND-parallel Prolog abstract architecture is to provide inference speeds significantly beyond those of sequential systems, while supporting Prolog semantics and preserving sequential performance and storage efficiency. This paper presents simulation results supporting these claims with special emphasis on memory performance on a two-level shared--memory multiprocessor organization. Several solutions to the cache coherency problem are analyzed. It is shown that RAP-WAM offers good locality and storage efficiency and that it can effectively take advantage of broadcast caches. It is argued that speeds in excess of 2 MLIPS on real applications exhibiting medium parallelism can be attained with current technology.

1 Introduction

The RAP-WAM execution model [10,11] is aimed at providing, through the use of parallelism, inference speeds to logic programs beyond those attainable in sequential systems, while supporting the conventional "don't know" non-deterministic semantics of logic languages. Of the various sources of parallelism present in Logic Programs [3] the RAP-WAM architecture exploits Goal Independence AND-parallelism [11], an extension of DeGroot's Restricted AND-parallelism [4] which provides backward execution semantics and improved execution graph expressions.[1] Sets of goals which are independent (i.e., which do not share any non-ground variables, determined by a combined compile-time, run-time analysis) are run in parallel. Parallelism can be programmed by the user by annotating the program with Conditional Graph Expressions (CGEs)[2] or it can be generated automatically by the compiler, through a combination of local and global (abstract interpretation-based) analysis [17] which often makes run-time independence checks unnecessary.

At the implementation level, the RAP-WAM architecture is designed to exploit both parallelism and advanced compiler technology. The techniques used for supporting parallel execution are extensions of those used in the Warren Abstract Machine (WAM)[15], which have already brought high inferencing speeds to sequential Prolog systems. Special attention is given to the preservation of WAM sequential performance and storage efficiency, and to the use of low overhead mechanisms for controlling parallel execution. Most of the WAM performance and storage optimizations are still supported during parallel execution. The CGE semantics has been integrated naturally into the WAM storage model in the form of specialized stack frames and storage areas which are used during parallel execution. Thus the default (sequential) model is that of a standard WAM exhibiting the same high sequential performance.

The RAP-WAM architecture can be viewed as a collection of abstract machines (workers) which cooperate in the execution of a program. Each of these abstract machines is similar to a standard WAM (featuring a complete set of registers and data areas, called a *Stack Set*), with the addition of a *Goal Stack* (used for on-demand scheduling), a *Message Buffer*, and two new types of stack frames: *Parcall Frames* and *Markers*. Parcall Frames coordinate and synchronize the execution of parallel goals both during forward execution and backtracking. Markers delimit *Stack Sections* (horizontal cuts through the Stack Set of a given abstract machine, corresponding to the execution of one parallel goal) and they implement the storage recovery mechanisms during backtracking[11]. In practice, the stack is divided into separate *Control* (Choice Point and Markers) and *Local* stacks (Environments) for reasons of locality and locking. Table 1 summarizes the types of objects allocated in these areas and their locality. Space limitations make a complete description of the RAP-WAM execution model impossible. The reader is referred to [11] for further details.

Frame type	area	WAM?	lock	locality
Envts./control	Stack	yes	no	Local
Envts./P. Vars.	Stack	yes	no	Global
Choice points	Stack	yes	no	Local
Heap	Heap	yes	no	Global
Trail entries	Trail	yes	no	Local
PDL entries	PDL	yes	no	Local
Parcall F./Local	Stack	no	no	Local
Parcall F./Global	Stack	no	no	Global
Parcall F./Counts	Stack	no	yes	Global
Markers	Stack	no	no	Local
Goal Frames	G. Stack	no	yes	Global
Messages	M. Buff.	no	yes	Global

Table 1: Characteristics of RAP-WAM Storage Objects

This paper presents simulation results for RAP-WAM supporting the claims of performance and efficiency. Although an evaluation of the implementation of the model on an existing shared-memory machine (Sequent) is currently also under way it only provides a single data point corresponding to a particular organization.[3] In addition, many statistics are very difficult

[1]The model is currently being extended to support OR-parallelism -using techniques similar to those proposed by other researchers, see for example [16,18] and their references- and a form of dependent AND-parallelism.

[2]CGEs offer Prolog syntax and permit conjunctive checks, thus lifting limitations in the expressions proposed by DeGroot: given "`f(X,Y,Z):- g(X,Y), h(Y,Z).`" the most natural annotation for this clause, that `g` and `h` can run in parallel if the terms in `X` and `Z` don't share variables and `Y` is bound to a ground term, can be expressed easily with CGEs ("`f(X,Y,Z):- (indep(X,Z), ground(Y) | g(X,Y) & h(Y,Z)).`") but is very difficult with DeGroot's expressions.

[3]Note also that the Balance model being used in this implementation uses write-through caches, which will be shown later in this paper to be not ideally suited for Parallel Prolog execution. Performance results from this implementation will be reported on elsewhere.

to gather from running hardware. Simulations can provide data over a wide range of architectural and organizational parameters and that is the approach taken in this study. Because high performance processing elements (PE's) are limited by available memory bandwidth (an even more important factor in parallel systems) this paper concentrates on memory performance.

The rest of the paper is organized as follows: results obtained from high-level simulations of the architecture are first summarized. A two-level shared-memory organization model and alternative solutions to the cache coherency problem are then proposed. Finally, RAP-WAM simulation results for the different coherency protocols proposed are presented and discussed.

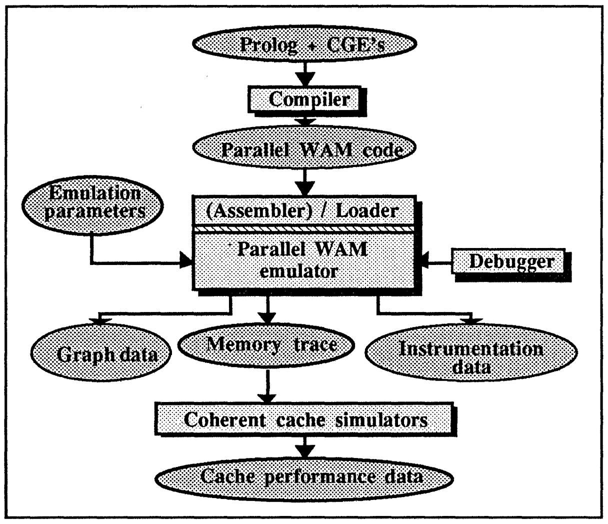


Figure 1: Simulation tools

2 Simulation Environment and High-level Results

A series of measurement tools have been built in order to evaluate the potential performance of the execution model and the associated architectural tradeoffs (Figure 1). Because the RAP-WAM model (as the WAM) is specified at a level above that of memory organization, simulations were first performed under the ideal assumption of a uniform, single shared-memory and no contention. The measurements were thereby made independent of the particular architectural organization on which the model is implemented. The emulator generated instrumentation data such as instruction frequencies, number of references classified by data areas, ratios of local vs. remote references, maximum amount of storage used per area, estimated timings, and speedups. Results from simulations at this level can be found in [12,11] and can be summarized as follows:

The overhead in the RAP-WAM model due to the management of parallelism is low: it has, for example, been observed to be in the order of 15% for up to 40 processors even for fine granularity cases (i.e., high overhead cases) such as that of the "deriv" benchmark, as shown in Figure 2. In this figure, *work* represents the number of references generated by all PEs while doing actual processing (i.e., not *waiting* or *idle*). Overhead, the difference between the work (references) done by RAP-WAM and that of WAM, is in Figure 2 the distance between the work curve and the "uniprocessor" line corresponding

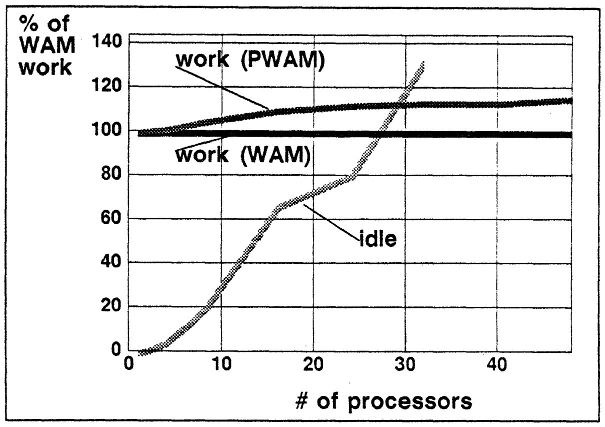


Figure 2: RAP-WAM Overheads for "deriv"

to WAM work. All data in this figure is presented as percentages of WAM work (executing the sequential version of the benchmark). Note that RAP-WAM work on 1 PE is very close to WAM work. Speedup (i.e. significantly faster execution than a high-performance sequential implementation -WAM- for similar performance PE's) is thus obtained even if the application exhibits only low levels of parallelism. The stack-based memory management approach[11] also appears to be very efficient recovering local storage upon procedure exit (with last call optimization) and all storage on backtracking as in the WAM.

Although these results are encouraging, practical memory organizations deviate from the ideal behavior assumed above and it is thus important to assess the effect of this deviation if realistic performance figures are to be obtained. This issue is addressed in the next sections by quantifying the effect of a particular memory organization with limited bandwidths, cache coherence maintenance overhead, etc.

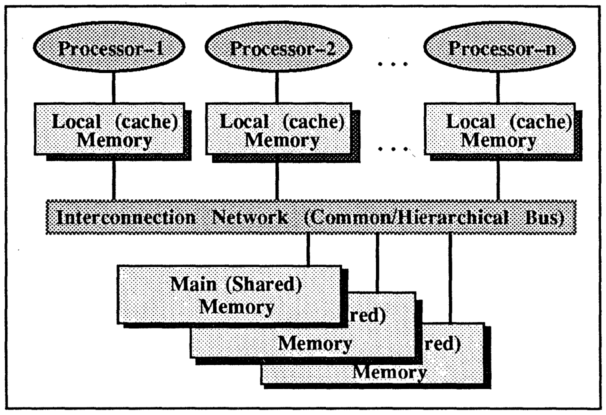


Figure 3: The Two-Level Shared-Memory Architecture Model

3 Two-Level Shared-Memory Results

Figure 3 shows a practical shared-memory system presenting a two-level structure where a local cache memory is located between each PE and the system bus. Such a hierarchical organization, characteristic of many current shared-memory multiprocessors, serves a dual purpose: first, in allowing faster execution because of the generally lower effective memory access time seen by a PE, essential in obtaining performance that is competitive with that of sequential systems. Second, in absorbing a (hopefully) significant part of the traffic to main memory which needs to go through the system bus, particularly important in shared-memory multiprocessors because the system bus is often the most significant bottleneck in the system. The locality of Prolog/WAM was shown by Tick[14]. In the next sections it is shown that Prolog/RAP-WAM also offers sufficient locality to take advantage of cache memories.

3.1 Cache Coherency

Except for simple buffers which hold only local data, most of the local memory designs used in conventional or special-purpose sequential machines for the implementation of logic programs (such as, for example, those used in [5] or those studied by Tick[14]) cannot be used directly in a parallel machine because of cache coherency problems. *Coherent caches* ensure that all the PEs in the system have a consistent view of the storage model. Although at certain times during the operation of RAP-WAM coherency is not *required*, it appears that ensuring coherency *continually* is easier than enforcing coherency only at specific points (and has the additional benefit of generality). Therefore, traditional coherent caches are considered in this study.

Historically, the first coherent caches[7], used a *write-through strategy*, where all write references were issued to both the local cache and shared memory, and copies residing on a cache other than the cache issuing the write request were invalidated. This *coherency protocol* is inexpensive in terms of hardware, but offers low performance because of excessive traffic on the system bus. Recently, a family of *fully distributed broadcast cache protocols* have been proposed and built [8,1,2] which are based on the ability of the cache organization to modify all copies of a cached item in all caches which share this item *in a single bus cycle*. Information is maintained for each cache block as to whether it is *private* or *shared*, making it possible to avoid coherency overheads for private blocks and implement write-back policies. Different designs differ essentially in the treatment of a write to a possibly shared block. A write-through broadcast strategy *updates* remote copies and possibly shared memory. A write-in broadcast strategy *invalidates* remote copies. Descriptions and measurements of the relative performance of various broadcast protocol attributes for conventional architectures are given in Archibald[1].

The broadcast protocol offers high performance at the expense of additional hardware. With the objective of reducing this expense by exploiting attributes of the RAP-WAM architecture, a *(firmware) controlled hybrid cache protocol* was developed. This scheme attempts to combine the efficiency of broadcast caches with the simplicity and low cost of a traditional write-through cache using information provided by the PE (in the form of tags, derived from the information in Table 1) as to the locality characteristics of each reference. The protocol is referred to as "hybrid" because based on these tags *potentially shared* (global) data is written-through and local data is copied back. An underlying tenet of the hybrid protocol is to avoid some of the complexity of broadcast caches by keeping shared memory consistent with local memory. The cost associated with this simplification is the traffic required to write-through to memory the write requests marked as global which are not actually shared. The gain with respect to the traditional write-through approach is that data marked as local is not written-through.

Parameter	deriv	tak	qsort	matrix
Instructions executed	33520	75254	237884	95349
References (RAP-WAM)	85477	178967	502717	96013
References (WAM)	82519	169599	499526	95357
Goals actually in //	97	263	97	24

Table 2: Statistics for the Benchmarks Used (8 processors)

cache size	E_{tr}	σ_{tr}	$(tr - E_{tr})/\sigma_{tr}$			
(words)	large bench		deriv	tak	qsort	mean
512	0.164	0.0626	1.1	-1.9	0.83	1.3
1024	0.108	0.0569	2.0	-1.1	1.6	1.6

Table 3: Fit of Small Benchmarks to Large Benchmarks

3.2 Simulations

In order to compare the performance of the various types of caches presented above, the RAP-WAM emulator was modified to generate a trace file of memory references (Figure 1). These references are marked with a *PE identifier*, a tag describing the particular storage area and object being accessed, and a read/write flag. All of the coherent cache models are simulated with the same parameterized multiprocessor cache simulator[14] which can be reconfigured to support the various consistency protocols. Caches are modeled as fully associative memories with perfect LRU replacement.

The results presented correspond to the execution of the following set of benchmarks: symbolic derivation ("deriv", which finds the symbolic derivative of a given arithmetic expression), Takeuchi ("tak", which computes Takeuchi's function), Quick-sort ("qsort", written using difference lists), and Matrix Multiplication ("matrix", a naive matrix multiplication program). Each benchmark was executed on relatively large input data. Table 2 shows some statistics regarding the benchmarks used, running on 8 PE's. Note that the number of references shows reasonable size. These benchmarks and their input data were chosen for several reasons: their small granularity (except for "matrix") provides a worst-case type of analysis with respect to parallelism management overhead. They also offer reasonable degrees of parallelism so that the parallel portion of the abstract machine is exercised. Also, their sequential memory referencing behavior and *locality* resemble those of much larger Prolog programs, such as the ones studied by Tick[14]: table 3 shows that the fit is quite good ensuring that the benchmarks exercise the sequential storage model (the foundation of the RAP-WAM storage model) in a reasonable, typical way.

Figure 4 shows the mean traffic ratios (as a function of *total* cache size and averaged over the four benchmarks) of the write-in broadcast, hybrid, and conventional write-through cache protocols, using four word lines. Caches of sizes 64, 128, and 256 words were simulated with no-write-allocate (a write miss does *not* fetch the corresponding block to cache). Caches of sizes 512 and 1024 words were simulated with write-allocate, except for hybrid caches which used no-write-allocate for 512 words. These selections were made on the basis of the policy which produced the lowest traffic. A clear result of the simulations is that no-write-allocate is best for small caches; however, miss ratio increases with no-write-allocate. Another result is that

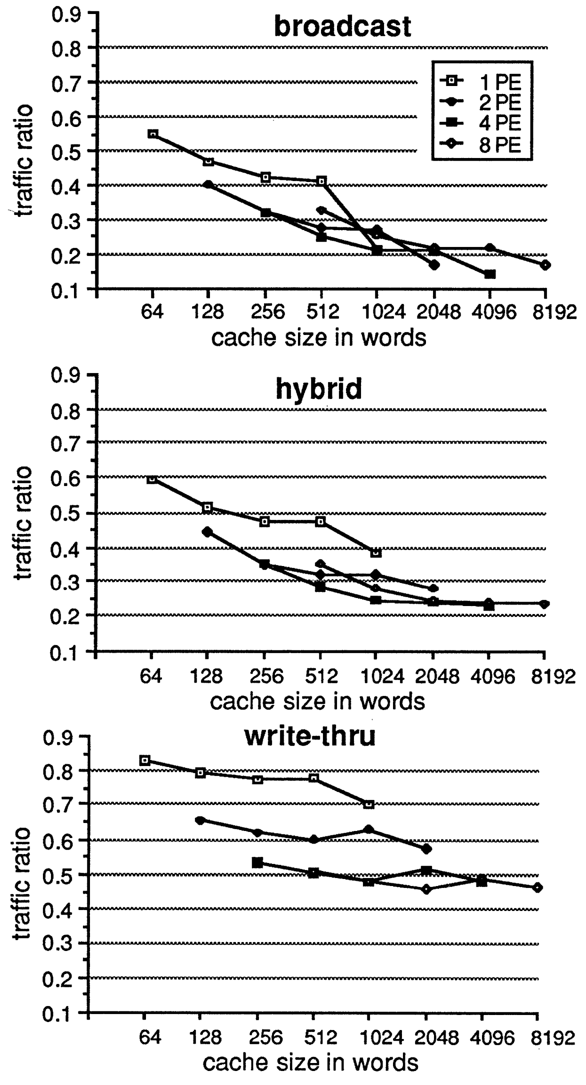


Figure 4: Traffic of Coherency Schemes

a more efficient replacement policy (e.g., copyback) produced lower traffic with write-allocate than a less efficient policy (e.g., hybrid) for the same cache size. The write-through broadcast cache statistics (not shown in Figure 4) are almost identical to those of the write-in broadcast cache, an indication that communication traffic in RAP-WAM is low.

A result seen from the curves is that the hybrid cache does quite well in reducing traffic, almost to the level of the copyback cache. The copyback cache does exceedingly well for 1024 word caches, and this trend is expected to continue with larger sizes, because the hybrid caches have already bottomed-out. The idiosyncrasies in the curves are due to the effects of averaging the benchmarks. Also, the advantageous effect (that of reducing memory traffic) of partitioning an algorithm's working set across several caches is seen to sometimes outweigh the increase in communication overheads. Lack of space makes it impossible to offer many simulation results. See [12] for more details on the benchmarks and simulations.

3.3 Discussion

As stated before, the hierarchical memory organization serves the dual purpose of lowering the effective memory access time and reducing the memory bandwidth requirement of a PE. According to the results of the simulations presented in the previous section, the hybrid cache generates an amount of traffic between that generated by the broadcast and conventional write-through caches. The broadcast schemes retain a (sometimes slight) advantage throughout the range of caches simulated.

It should be noted that these results measure performance only in terms of traffic ratio. For example, the simulation data shows that eight PEs with write-in broadcast caches (of 128 words or greater) generate a traffic ratio of less than 0.3 (the hybrid cache is also close to this performance); i.e. more than 70% of the traffic generated by the processors is captured in the local memories and will not appear on the bus. However, in order to accurately estimate the actual performance of a multiprocessor the time penalty to access shared memory due to contention must also be analyzed. Although beyond the scope of this paper a queueing model for this purpose is proposed in[14]. Results presented therein for RAP-WAM execution show that with a relatively fast bus and an interleaved memory shared memory efficiency can be high.

It is of obvious interest, if only to stimulate further research, to speculate about the potential performance levels attainable given the results presented in the previous sections. Even current low- to medium-cost shared-memory systems offer high PE to memory bandwidths by implementing multiple or overlapped busses and interleaved memories. This makes it reasonable to predict that speeds in the order of 2 million *application*[4] inferences per second are possible on shared-memory multiprocessors built using current technology.[5] A "back of the envelope" calculation, in order to justify this claim and based on the results obtained from the present and previous studies can be made as follows: studies of large Prolog benchmarks show that in the average 15 (WAM or RAP-WAM) instructions are executed per actual inference and that each instruction averages 3 (word) references. This represents 45 words/LI, or 180 bytes/LI for a 32 bit word size. Therefore, a system executing at a speed of 2 MLIPS would require a cumulative memory bandwidth of 360 Mbytes/sec. If the caches are able to capture 70% of this traffic, only 108 Mbytes/sec have to be delivered by the bus/memory system, a performance which is perfectly achievable using current off-the-shelf technology.[6]

4 Conclusions

The paper has presented memory referencing characteristics of a parallel logic programming architecture, RAP-WAM, based on Independent/Restricted AND-parallel execution of Prolog, and

[4]"Application" inferences refer to inference steps of the average size found in large Prolog programs, i.e. in the order of 15 WAM instructions. This results in much lower but more realistic figures than those obtained using the conventional "LIPS" measurement based on "naive reverse."

[5]Note that the Japanese FGCS Project is also predicting similar inferencing speeds for the PIM[9].

[6]These conclusions, although resulting from more detailed simulations than those presented in a related study by Fagin[6], are in disagreement with Fagin's results and his contention that Prolog programs cannot effectively make use of multiprocessing. The discrepancies are probably due to differences in the execution models used and to the small size of the benchmarks/data simulated by Fagin. They do agree, however, with those of Lin [13].

its behavior and potential performance on shared-memory multiprocessor organizations. The measurements presented here indicate that RAP-WAM is well-suited to high performance execution on tightly-coupled shared-memory multiprocessors, from cost-effective small-scale systems to higher-performance medium-sized systems. It has been argued that actual speeds of 2 Million *application* inferences per second are possible with currently available technology for applications which exhibit medium degrees of parallelism. It has been shown that the architecture offers high memory referencing locality so that it can take advantage of two-level memory organizations. The memory referencing study included comparison of cache coherency protocols and the "broadcast" and "hybrid" protocols were shown to offer superior performance to write-through mechanisms, present in some multiprocessors.

Because the memory organizations studied are characteristic of many current and next-generation multiprocessors, it is argued that the results obtained are relevant to the estimation of the performance of AND-parallel Prolog/RAP-WAM on them and also to determining the advantages and shortcomings of such machines in the parallel implementation of other don't-know non-deterministic logic programming languages and models. In addition, the results can also be used as a guideline in the design of small to medium-sized special purpose multiprocessors. Although the goal of small to medium systems may seem rather unambitious, it is important to have evidence of actual speedups at these levels before attempting the design of large-scale systems. In the words of the adage, "Walk before you run..."

5 Acknowledgements

Thanks to Richard Warren for his comments on earlier drafts of this paper, MCC for its support, and ICOT for supplying facilities with which a portion of this paper was formated. Evan Tick was also supported by an IBM Graduate Fellowship and by NASA-Ames (NCA2-1R745-406).

References

[1] J. Archibald. *High Performance Cache Coherence Protocols For Shared-Bus Multiprocessors.* Technical Report 86-06-02, University of Washington, Seattle, WA 98195, June 1986.

[2] P. Bitar and A. M. Despain. Multiprocessor Cache Synchronization. In *13th Int. Symp. on Comp. Arch.*, pages 424–433, June 1986.

[3] J. S. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Norwell, MA 02061, 1987.

[4] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478, November 1984.

[5] T. P. Dobry et. al. Performance Studies of a Prolog Machine Architecture. In *12th Int. Symp. on Comp. Arch.*, pages 180–190, December 1985.

[6] B. Fagin and A. Despain. Performance Studies of a Parallel Prolog Architecture. In *14th Annual International Symposium on Computer Architecture*, pages 108–116, IEEE Computer Society, June 1987.

[7] D. H. Gibson. Considerations in Block-Oriented Systems Design. In *AFIPS Conference Proceedings*, pages 75–80, Spring Joint Computer Conference, Academic Press, April 1967.

[8] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *10th Annual International Symposium on Computer Architecture*, pages 124–131, IEEE Computer Society, 1983.

[9] A. Goto. Parallel Inference Machine Research in FGCS Project. In *Proceedings of the First Japan-U.S. AI Symposium*, pages 21–36, December 1987.

[10] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Proceedings of the Third International Conference on Logic Programming*, pages 25–40, Springer-Verlag, 1986.

[11] M. V. Hermenegildo. *Independent AND-Parallel Prolog and its Architecture.* Kluwer Academic Publishers, Norwell, MA 02061, 1988.

[12] M. V. Hermenegildo and E. Tick. *Memory Performance of AND-Parallel Prolog on Shared-Memory Architectures.* Technical Report PP-036-88, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, January 1988.

[13] Y.-J. Lin. *A Parallel Implementation of Logic Programs.* PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.

[14] E. Tick. *Studies In Prolog Architectures.* PhD thesis, Stanford University, Stanford, CA 94305, June 1987.

[15] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Report 309, SRI International, 1983.

[16] D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *1987 Symposium on Logic Programming*, pages 92–102, IEEE Computer Society, August 1987.

[17] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, August 1988.

[18] H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Symp. of Logic Prog.*, pages 436–448, August 1987.

COMPILING ENUMERATE-AND-FILTER PROGRAMS FOR EFFICIENT EXECUTION UNDER COMMITTED-CHOICE AND-PARALLELISM [1]

Arvind K. Bansal and Leon S. Sterling
Dept. of Computer Engineering and Science,
Case Western Reserve University,
Cleveland, OH. 44106 (USA)

Abstract
This paper presents a technique and algorithms for compilation of *enumerate-and-filter logic programs*, for efficient execution under committed-choice AND-parallel logic programming languages. The compilation technique preserves the integration of OR-parallelism, AND-parallelism and stream parallelism present in enumerate-and-filter programs. Algorithms are demonstrated by compiling enumerate-and-filter programs in Flat Concurrent Prolog. Compilation of enumerate-and-filter construct improves the execution time by an order of magnitude. Comparison of the sequential Prolog version of compiled enumerate-and-filter programs and *setof* construct in Prolog demonstrates that efficiency is achieved without any extra run time overhead.

Keywords AND-parallelism, committed-choice, compilation, enumerate-and-filter, generate-and-test, logic program, OR-parallelism, stream parallelism.

1. Introduction

In a previous paper [1] we introduced the enumerate-and-filter paradigm to execute generate-and-test logic programs using committed-choice AND-parallelism. The paradigm enumerates possible candidate solutions using OR-parallel set enumeration, spawns a tester process for each candidate solution, and collects the solutions for which the tester process terminates successfully. The transformation integrates almost full OR-parallelism, AND-parallelism and stream parallelism. OR-parallelism was simulated using stream AND-parallelism in enumerating the set of solutions and in operationally nondeterministic testers.

The OR-parallel *setof* uses explicit copying to spawn a different copy of the rest of the conjunctive subgoals for every possible enumerated solution. The filter process uses metacalls for spawning testers. The stream operations also make explicit copies. The use of metacalls and explicit copying is expensive in terms of execution.

This paper presents a general compilation technique which removes the metacalls and explicit copying. Algorithms are presented and demonstrated by compiling enumerate-and-filter programs in Flat Concurrent Prolog (FCP) a committed-choice AND-parallel language [8] taken as an example language to express parallelism. Our techniques are more general and can be adapted for other committed-choice AND-parallel languages. The compiled programs will run efficiently under non Von-Neumann architectures.

The structure of the paper is as follows. The next section gives the basic concepts, definitions and a brief overview of generate-and-test and enumerate-and-filter paradigm. Section 3 discusses the compilation technique, algorithms for compilation and statistics. The last two sections compare related work and discuss conclusions respectively.

2. Basic Concepts

2.1 Definitions

This paper assumes a familiarity with logic programming and Prolog [9] and committed-choice AND-parallel logic programming [8], [1]. We introduce some new terminology in this section.

The definitions of input and output variables are standard and present in [4], [5]. An *Input term* is a term with atleast one input variable, and no output variables. An *Output term* is a term containing an output variable. Given the information about input and output variables at top level goal abstract interpretation using type expressions determines input and output variables for the predicates in the program [3].

A *computation* of a logic program is a sequence of reductions (or resolution steps) from an initial query using clauses from the program. At each stage the goal to be reduced is determined by some computation rule. For this paper the computation rule of Prolog, namely choosing the leftmost goal, is assumed unless otherwise specified. A computation is *successful* if the empty goal is reached. In this case, the binding of the variables in the initial query constitutes a solution. A computation is *failed* if there is no clause to reduce the chosen goal.

A program is *rigidly deterministic* with respect to a query if every computation starting from the query chooses a unique clause to resolve the goal selected by the computation rule. There is only one successful (or failed) computation.

A program is *loosely deterministic* with respect to a query if there is at most one successful computation for the query, though there may be several failed computations.

A program is *don't care nondeterministic* for a query if whenever there are several successful computations for the query yielding same solution.

A program is *pluralistic* for a query if there are multiple successful computations with different solutions.

2.2 Classifying Generate-and-test Programs

Generate-and-test programs have at least one clause which has at least one subgoal *G* which generates multiple values for at least one variable *V*. The subgoal *G* is followed by at least one subgoal *T* with no output variable and sharing the variable *V*. The subgoal G is called the generator and the subgoal T is called the tester. Generate-and-test programs are an important subclass of pluralistic programs.

Generate-and-test programs are classified to three classes

1. A *simple generate-and-test program* has generate-and-test goal pair in nonrecursive clauses.

2. In a *recursively embedded generate-and-test program* the generate-and-test pair is embedded in a recursive clause to test the partial solutions for eager pruning of the search space. The generator and tester are identified using abstract interpretation [3]. The intermediate solution is the final solution if the input term is completely consumed or the intermediate solution meets the final constraint. The N queens program (Figure 1) is a typical example . The predicates *select/3* and *attack/2* act as generator and tester respectively.

3. In *deeply intertwined generate-and-test programs*, the same predicate is a generator if certain variables are uninstantiated and tester if the variables are instantiated. Unification is used implicitly for instantiating the uninstantiated variables

[1] Supported in part by Center of Automation and Intelligent Systems Research, CWRU, Cleveland, OH 44106

and testing the instantiated variables.

2.3 Enumerate-and-filter Paradigm

Committed-choice AND-parallel model lacks the capability of multiple solutions due to single clause commitment and absence of backtracking. Multiple solutions are incorporated in committed-choice And-parallel models using the enumerate-and-filter paradigm [1]. The basic relation for the idiom is

enumerate_and_filter(Term, Enum, Test, Stream).

An enumerator produces the set of possible solutions which are filtered by a tester which can be either rigidly deterministic, loosely deterministic, don't care non deterministic predicate. The enumerate-and-filter paradigm executes pluralistic programs. Generate-and-test programs are subset of pluralistic programs.

Sets of solutions are represented as streams allowing the exploitation of stream parallelism. The stream operations required for transformation are merging two streams, mixing structures in a single stream or multiple streams to get a new-stream, filtering streams, closing streams, testing for empty stream [1]. The predicate *mix_stream/4* is used to mix structures of various streams to get a new stream. The basic relation for *mix_stream/4* is *mix_stream(In, Out, Instrms, Outstrm)*.

```
queens(N, Qs) :- one_to_n(N, Ns), queens(Ns,[ ],Qs).
queens(U, S, Qs) :-
    select(Q, U, U1), \+ attack(Q, S), queens(U1, [Q | S], Qs).
queens([ ], Qs, Qs).
select(X, [X | Xs], Xs).
select(X, [Y | Ys], [Y | Zs]) :- select(X, Ys, Zs).
attack(X, Ys) :- attack(X, 1, Ys).
attack(X, N, [Y | Ys]) :- X is Y + N.
attack(X, N, [Y | Ys]) :- X is Y - N.
attack(X, N, [Y | Ys]) :- N1 is N + 1, attack(X, N1, Ys).
```

Figure 1: N Queens program in Prolog

```
queens(N, Qs) :-
    one_to_n(N, Ns), queens1([(Ns?, [])], Qs).
queens1([], Qs).
queens1([I | Is], Qs) :-
    otherwise | queens2(I, Qs), queens1(Is?, Qs).
queens2(([], S), Qs) :- result_writer(S, Qs).
queens2((U, S), Qs) :- otherwise |
    enumerate_and_filter((U1, Q),
        select(Q, U, U1), not_attack(Q, S), R),
    mix_stream((A, B), (A, [B | S]), R?, R1),
    queens1(R1?, Qs).
not_attack(X, [], true).
not_attack(X, Ys, R) :-
    otherwise | attack(X, 1, Ys, R1), flip_result(R1?, R).
attack(X, _, [], false).
attack(X, N, Ys, R) :- otherwise |
    attack1(X, N, Ys, R1),attack2(X, N, Ys, R2),
    attack3(X, N, Ys, R3), or_solution([R1, R2, R3], R).
attack1(X, N, [Y | Ys], R) :- plus(Y, N, Y1), eq(X, Y1?, R).
attack2(X, N, [Y | Ys], R) :- diff(Y, N, Y1), eq(X, Y1?, R).
attack3(X, N, [Y | Ys], R) :- plus(N, 1, N1), attack(X, N1?, Ys, R).
```

Figure 2: Transformed N queens program in FCP

3. Compiling Enumerate-and-filter

The OR-parallel *setof* construct used to realize enumerator [2] explicitly copies all the conjunctive goals occurring after the current goal being evaluated for every possible enumerated solution. The filter construct makes explicit metacalls to spawn a tester process for every possible enumerated solution. Metacalls and explicit copying are expensive in terms of execution.

3.1 Compiling Filters

The filter process takes every solution from the stream generated by the enumerator and makes a metacall for the tester process. Compiling the process is straightforward. A new recursive process is started which spawns the tester for every possible solution. The tester returns the value of result variable *Res* depending upon the outcome. A conditional writer process *conditional_write/3* is spawned for every enumerated solution which writes on the global solution stream if the value of the result variable *Res* is *true*. The first element of the stream contains the complete structure information for implicit unification and variable generation (refer [2] for the formal algorithm).

```
filter_not_attack([ ],_,_).
filter_not_attack([(U, S) | Xs], Ys, Qs) :-
    otherwise | not_attack(S?, Ys, Res),
    conditional_writer(Res?, (U?, S?), Qs),
    filter_not_attack(Xs?, Ys?, Qs).
conditional_writer(false, _, _).
conditional_writer(true, X, Qs) :- result_writer(X, Qs).
not_attack(S, Ys, Res) :- see Figure 2
```

Figure 3: Compiled filter for *not_attack/3* in FCP

3.2 Compiling Stream Operations

The predicate *mix_stream/4* is the only stream operation which makes use of explicit copying and unification. The predicate *mix_stream/4* is compiled to the predicate *mix_stream/3*

```
mix_stream(_,[ ],[ ]).
mix_stream(Intrm, [Intuple | Ins], [Outtuple | Outs]) :-
    mix_stream(Intrm, Ins?, Outs).
```

The first argument in *mix_stream/3* is the input term which is not present in the input streams. This input term is mixed with first element of the input streams as specified in the second argument to give the first element the output stream as specified in the third argument. The structures of the first element in the input streams and the output stream are made explicit in the second and third arguments to make use of implicit unification and variable generation.

3.3 Compiling Enumerators

The OR-parallel setof construct needs following operation to enumerate all the possible values.

(1) Finding the clauses having same predicate-name and arity concurrently. (2) Making as many copies of the call as there are number of clauses. (3) Unifying each copy of the call with one of the clauses. (4) Filtering the successful clauses and finding out the solution for successful clause matches concurrently. (5) building partial solutions bottom-up for recursive clauses. (6) Matching the union of the shared terms and output terms with their solutions for each successful clause. The successful solutions are filtered to the final solution stream. (7) Merging the solution streams for each successful clause.

The generators and testers in simple for generate-and-test and recursively embedded generate-and-test programs are determined using program analysis. Input variables and output variables are identified. Such programs are transformed by compiling each clause of the enumerator to a single clause procedure which returns a stream of solutions. These solution streams are merged. For recursive clauses, the solution is built bottom-up. We give the algorithm in Figure 6.

A compiled version of the enumerator *select/2* in the N-queens program is given in Figure 2. The top level procedure *select/2* has two single clause procedures (SCPs) corresponding to the two clauses in *select* in Figure 2. The last subgoal *merge/3* in *select/2* merges two solution streams *S1* and *S2*. The recursive predicate *select_clause2* builds the solution bottom-up.

```
select(Qs, S) :-
    select_clause1(Qs?, S1), select_clause2(Qs?, S2),
    merge(S1?, S2?, S).
select_clause1([ ],[ ]).
select_clause1([Q | Qs], [(Qs, Q)]).
select_clause2([ ],[ ]).
select_clause2([Q | Qs], S) :-
    select(Qs?, R1),mix_stream(Q?, R1?, S).
mix_stream(Q, [(X, Y) | R], [([Q | X], Y) | R1]) :-
    mix_stream(Q, R?, R1).
mix_stream(_,[ ],[ ]).
```

Figure 4: Compiling *select/2* of N queens program in FCP

```
queens/2 & queens1/2 as given in Figure 2
queens2(([ ],S),Qs) :- result_writer(S, Qs).
queens2((U, S), Qs) :-
    otherwise | select(U?, R),
    filter_not_attack(R?, S?, R1),
    mix_stream_queens2(S?, R1?, I_stream),
    queens1(I_stream?, Qs).
mix_stream_queens2(S, [(U, Q) | Xs], [(U, [Q | S]) | Ys]) :-
    mix_stream_queens2(S, Xs?, Ys).
mix_stream_queens2(_,[ ],[ ]).
select/3 as given in Figure 4.
filter_not_attack/3 as given in Figure 3
```

Figure 5: Compiled version of N queens program in FCP

```
Algorithm compile static-enumerator
Input: Enumerator, Output: Compiled enumerator
Procedure compile recursive-clause(top level procedure)
Begin
  Add a subgoal invoking top level procedure;
  If there is any output list in the recursive clause whose first
  element is instantiated by non pluralistic subgoal in the body
  of recursive clauseThen
  Add a subgoal mix_stream(First,Rest_stream, Result);
  Else if the first element is instantiated by a pluralistic subgoal
  in the body of the recursive clause Then Add a subgoal
  recursive_mix_stream(First_stream, Rest_stream,Result);
End;
Begin
  Rename each clause of the enumerator to make them a single
  clause procedure(SCP). Each SCP has two arguments (1) a tu-
  ple of input terms (2) a result stream of tuples formed by union
  of shared terms and output terms;
  Create a top level procedure say p/2 with subgoals correspond-
  ing to these SCPs.;
  call procedure compile recursive-clause(p/2) for the SCP corre-
  sponding to recursive clause;
  In the body of this top level procedure p/2, add a subgoal merge
  with arity n+1 (n = no. of SCPs), to merge the individual
  streams generated by SCPs.
End.
```

Figure 6: Compiling a statically inferrable enumerator

The dynamic enumerator corresponding to generator/tester is compiled using a run time test to determine the instantiation state of the variables whose instantiation states change dynamically. Such variables are determined using mode-analysis with type-expressions [3]. Each nonrecursive clause is expanded into 2^n (n = number of such variables) mutually exclusive sub single clause procedures to take care of all the combinatorial possibilities. This check for *uninstantiated* or *ground* is done in the guard. For the test *ground(Variable)* an extra subgoal is added in the guard to match the generated value and the value of the variable. The compilation of recursive clauses is similar to recursive clauses for static enumerators. An algorithm is given in Figure 7.

```
Algorithm Compile dynamic-enumerator;
Input: A dynamic enumerator and set of variables whose
instantiation state changes dynamically;
Output: A compiled dynamic enumerator;
Begin
  Each single clause procedure (SCP) has three arguments (1) a
  tuple representing the union of shared and output terms,(2) a
  tuple of variables whose instantiation state has to be tested, (3)
  the stream of shared and output terms;
  Case SCP of
    Non-recursive or base-case:
      Explode each SCP into $2^n$ SSCPs having in its guard
      test for every variable X which changes the instantiation
      state dynamically, where n is number of such variables;
      For each SSCP find out the subgoals in the guard. For
      the test ground(X) treat X as input variable. For the
      test Var(X) treat X as output variable;
      Compile the body of SSCP using the algorithm to com-
      pile static enumerator (Figure 6) with new input and
      output variable information;
    Recursive:
      Compile the clause using procedure compile recursive
      clause as given in Figure 6;
End.
```

Figure 7: Compiling dynamic enumerator (generator/tester)

Consider the predicate remove/3

```
remove(X, [X | Xs], Xs).
remove(X, [Q | Qs], [Q | Xs]) :-
    remove(X, Qs, Xs).
```

The nonrecursive clause has two variables *X* and *Xs* which can be either ground or uninstantiated depending upon the dataflow. Hence *remove_clause1/3* is exploded into 4 clauses as shown below

```
remove_clause1((X, Xs), [Q | Qs], Z) :-
    ground(X), ground(Xs), (X?, Xs?) = = (Q?, Qs?) |
    Z = [(Q, Qs)].
remove_clause1((X, Xs), [Q | Qs], Z) :-
    ground(X), var(Xs), X? = = Q? | Z = [(Q, Qs)].
remove_clause1((X, Xs), [Q | Qs], [(Q, Qs)]) :-
    ground(Xs), var(X), Xs? = = Qs? | Z = [(Q, Qs)].
remove_clause1((X, Xs), [Q | Qs], [(Q, Qs)]) :-
    var(X), var(Xs) | true.
remove_clause1(_,_,[ ]) :- otherwise | true.
```

The guard of each clause tests for one of the four possibilities. If *X* or *Xs* is *ground* then their value is matched against the generated values. Binding of the element in the output stream is done only after the values match.

4. Results And Related Work

We executed both transformed and compiled 4-queens program on an FCP simulator (Logix version 1.1) on Vax 11/780 . The execution time was 13900 msec. and 520 msec. respectively, an order of magnitude improvement.

Ueda [10], [11] presents a method for making exhaustive search programs deterministic. In his scheme OR-parallelism is transformed to AND-parallelism and conjunctive goals are solved AND-sequentially. The scheme [11] uses subcontinuations to invoke the testers as soon as partial solution is generated by the generator. The number of shared terms between generators and testers is restricted. No conjunctive generators using the same variables are allowed, and testers are restricted to the predicates with input variables.

The Enumerate-and-filter paradigm integrates stream parallelism, AND-parallelism and almost full OR-parallelism. OR-parallelism is simulated using AND-parallelism in enumerating the set of multiple solutions and in loosely deterministic testers. Filter process spawns a tester as soon as any enumerated partial solution is available. In a recursively embedded enumerate-and-filter program filtering occurs at different levels concurrently. Pluralistic programs are executed which form a superset of generate-and-test programs. There are no restriction on the number of shared terms between generator and tester. A stream of tuples of shared terms is passed between generator and tester. Conjunctive generators with shared terms are allowed. The implicit pipelining in set enumeration prunes away the solutions with conflicting bindings.

Comparison of our scheme with Ueda's scheme (Program 5 of [11] adapted to run in FCP) as shown in Table 1 indicates that our scheme runs 10 % faster under FCP simulation on single processor. We believe this gap will increase on a multiprocessor architecture capable of exploiting pipelining and AND-parallelism. Dedicated processing elements may speed up stream operations like *mix_stream* and *merge*.

	Ueda's Scheme	*Our Scheme*
N	Time/Reductions	Time/Reductions
	msec/FCP(Prolog)	msec/FCP(Prolog)
4	580/ 933(563)	520/ 812(361)
5	2350/3618(2208)	2130/ 3287(1122)
6	9230/14388(7899)	8810/12839(4214)

Table 1: comparing FCP version of N-queens

Comparison of Prolog version of our program, Ueda's program [11] and *setof* shows that **(1)** all the schemes are comparable (see table 2) **(2)** there is no runtime overhead in our scheme. **(3)** number of reductions in our scheme is 40 % less. **(4)** our scheme is faster on Cprolog. The little variation in execution time using Sicstus and Quintus compilers is attributed to optimizations used for *setof* and compiler architecture supporting continuation. Comparison with other related research is present in [2] and has been omitted due to space limitations.

N	Unit	*Cprolog*	*Sicstus*	*Quintus*
		Vax 11/780	*Sun 2/50*	*Apollo DN300*
4	msec	335/380/321	61/74/72	83/66/74
5	msec	1470/1480/1292	214/196/203	308/245/251
6	msec	4930/5640/5060	477/616/689	573/850/912
7	sec	23.6/24.8/22.0	2.0/2.5/2.7	2.9/3.6/3.7
8	sec	108/115/103	-/-/-	11.3/20.3/20.9

Table 2: Comparing execution time of setof/Ueda/Our Scheme for N-queens in Prolog

5. Conclusions

We have developed techniques and algorithms to compile enumerate-and-filter programs for efficient execution under committed-choice AND-parallelism by removing explicit copying and use of metacalls. The compiled code preserves the declarative style of programming present in logic programming language and the integration of Or-parallelism, AND-parallelism and stream parallelism achieved by our transformation scheme [1]. The generate-and-test programs execute efficiently under committed-choice AND-parallelism when compilation techniques are combined with transformation techniques. Compilation increases the execution speed by an order of magnitude. Compiled enumerate-and-filter programs would run more efficiently than Ueda's continuation based scheme on non Von-Neumann architecture capable of exploiting AND-parallelism and pipelining. Compiled enumerate-and-filter programs are as efficient as *setof* construct in Prolog and Ueda's scheme under sequential execution showing that there is no extra runtime overhead.

References

[1] A. K. Bansal and L. S. Sterling, "On Source-to-Source Transfromation Of Sequential Logic Programs To AND-parallelism", *Proceedings International Conference On Parallel Processing*, (August, 1987), pp. 795 - 802.

[2] A. K. Bansal and L. S. Sterling, *Compiling Generate-and-test Programs to Committed-choice And-parallelism*, Dept. Of Computer Science, C W R U, Cleveland, OH 44106, USA, CES 87-13,(September, 1987).

[3] A. K. Bansal and L. S. Sterling, *Abstract Interpretation of Logic Programs for Transformation to Committed-choice And-parallelism*, Dept. Of Computer Science, C W R U, Cleveland, OH 44106, USA, CES 87-14, (October, 1987)

[4] S. Gregory, *Parallel Logic Programming in Parlog, The language and its implementation*, Addison Wesley, (1987)

[5] M. Codish, and E. Y. Shapiro, "Compiling OR-parallelism into AND-Parallelism", *International Conference on Logic Programming*, London, (July, 1986), pp. 283-297.

[6] J. Crammond, " An Execution Model for Committed-Choice Non-Deterministic Languages", *Proc. IEEE Symposium on Logic Programming*, Salt Lake City, Utah, USA, (August, 1986), pp. 148-158.

[7] A. Mycroft, R. A. O'keefe, " A Polymorphic Type System for Prolog", *Artificial Intelligence 23*, (1984), pp. 295 - 307.

[8] E. Y. Shapiro, *Concurrent Prolog - Collected Papers*, MIT Press, (1987), pp. 27-83.

[9] L. Sterling, and E. Shapiro,*The Art Of Prolog*, MIT Press, 1986.

[10] K. Ueda, "Making Exhaustive Search Programs Deterministic", *Proc. 3rd International Conference on Logic Programming*, London, UK, (July, 1986), pp. 270 - 282.

[11] K. Ueda, "Making Exhaustive Search Programs Deterministic II", *Proc. 4th International Conference on Logic Programming*, Melbourn, Australia, (May, 1987), pp. 356 - 375.

INDEPENDENT PARTITIONING OF ALGORITHMS WITH UNIFORM DEPENDENCIES

Weijia Shang and Jose A. B. Fortes
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

Abstract: An algorithm can be modeled as an *index set* and a set of *dependence vectors*. Each index vector in the index set indexes a computation of the algorithm. If the execution of a computation depends on the execution of another computation, then this dependency is represented as the difference between the index vectors of the computations. The *dependence matrix* corresponds to a matrix where each column is a dependence vector. An *independent partition* of the index set is such that there are no dependencies between computations that belong to different blocks of the partition. This paper considers uniform dependence algorithms with any arbitrary kind of index sets and proposes a computationally inexpensive method to find independent partitions of the index set. For most algorithms, this partition is maximal and the proposed method outperforms previously proposed approaches in terms of computational complexity and/or optimality. Also, lower bounds and upper bounds of the cardinality of the maximal independent partitions are given. For some algorithms it is shown that the cardinality of the maximal partition is equal to the determinant of one of the submatrices of the dependence matrix. Therefore, the value of this determinant determines the partitionability of the algorithm.

1. INTRODUCTION

Parallel processing holds the potential for computational speeds that surpass by far those achievable by technological advances in sequential computers. This potential is predicated on two often conflicting assumptions, namely, that many computations can take place concurrently and that the time spent in data exchanges between these computations is small. In order to meet these assumptions, algorithms and/or programs must be partitioned into computational blocks that can execute in parallel and have communication requirements efficiently supported by the target parallel computer. Ideally, it may be desirable to identify, if at all possible, the independent computational blocks of a program, i.e., those that require no data communication between them. This paper describes a practical and computationally inexpensive approach to achieve this goal. It is based on a sound mathematical framework which yields optimal results for a meaningful class of algorithms and it outperforms approaches proposed in extant work.

The identification of a possible partition of an algorithm or program can be done by the user, by the analysis phase of an optimizing compiler or by the machine at run time [4]. The techniques proposed in this paper, while usable by a patient and dedicated programmer, are best suited for an optimizing compiler. They address the specific problem of identifying *independent partitions* of an algorithm with goals that are similar to those of the early works of D.A. Padua [9] and J. Peir, D. Gajski and R. Cytron [11], [12], [13]. The focus of these efforts is on the optimization of programs consisting mainly of nested loops with regular data dependencies. The techniques proposed in those papers are intended to complement many other tools for the analysis and restructuring of sequential programs for execution in multiprocessing machines [1], [7], [10], [14], [15]. A related potential application of partitioning techniques is in the design of algorithmically specialized concurrent VLSI architectures [8].

This research was supported in part by the National Science Foundation under Grant DC1-8419745 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract No. 00014-85-k-0588.

In this paper, nested loop programs with regular data dependencies are modeled as *uniform dependence algorithms* which resemble the uniform recurrence equations considered in [6] and the linear recurrences of [11]. Data dependencies are represented as dependence vectors (with as many entries as the number of nested loops) that describe the distance between dependent computations in terms of loop indices (the vectors are called dependence distance vectors in [11] and are also considered in [15] and [2] in a complemented form). Dependence vectors are collected in a matrix, the *dependence matrix*, which is used in this paper and in [9], [11] and [13] to identify independent partitions as briefly described in the following paragraphs.

The *greatest common divisor method* [9], [11] considers, for each row of the dependence matrix, the greatest common divisor of the entries in that row. The resulting greatest common divisors are used to partition the iteration space of the program (also called the index set) and the cardinality of the resulting partition is the product of the greatest common divisors. In addition, an "alignment" method is provided in [9] which allows in some cases the transformation of dependencies so that the value of the greatest common divisors is increased. For a given set of dependencies, this approach yields a unique independent partition which is not necessarily optimal. In some cases, when all of the greatest common divisors equal unity, the number of the blocks in the partition is one, i.e., the whole program.

In the *minimum distance method* [11], [13], the dependence matrix is transformed into an upper triangular matrix which is then used to identify an independent partition. For some algorithms the cardinality of the partition is the product of the diagonal elements of the upper triangular matrix. This approach yields partitions which are better than those obtained through the greatest common divisor method. However, the computational complexity of this method is high (though affordable according to [11]) and the optimality is not guaranteed.

In the method proposed in this paper, a set of vectors defined later in Section 3 is derived from the dependence matrix. These vectors are used to find independent partitions of uniform dependence algorithms with any arbitrary kind of index set. The block to which a given index point belongs to can be identified by simply computing the dot products of each of the vectors by the index point. For some algorithms, the cardinality of the partition is equal to the absolute value of the determinant of a submatrix of the dependence matrix and for a meaningful class of algorithms, the partition obtained is maximal. A comparison of the method proposed in this paper with the minimum distance method is provided in Section 5.

The organization of this paper is as follows. Section 2 presents basic definitions and notation. In Section 3, partitioning vectors are defined and three types of independent algorithm partition by these partitioning vectors are derived. In Section 4, a procedure to find an independent algorithm partition is presented and sufficient conditions for the resulting partition to be maximal are discussed. Section 5 compares the method proposed in this paper to the minimum distance method. Finally, Section 6 concludes this paper and points out some open problems.

2. BASIC DEFINITIONS AND NOTATIONS

Throughout this paper, *sets*, *matrices* and *row vectors* are denoted by capital letters, *column vectors* are represented by lower case symbols with an overbar and *scalars* correspond to lower case letters. The *transposes* of a vector $\bar{v}$ and a matrix M are denoted $\bar{v}^T$ and M^T, respectively. The symbol E_i denotes the row vector whose entries are all zeros except that the ith entry is equal to unity. The vector $\bar{1}$ (or $\bar{0}$) denotes the row vector or column vector whose entries are all ones (or zeroes). The dimensions of $\bar{1}$ and $\bar{0}$ and whether they denote row or column vectors are implied by the context in which they are used. The vector space spanned by a set of vectors $S=\{\bar{v}_1, \bar{v}_2, ..., \bar{v}_k\}$ is denoted $sp\{\bar{v}_1, \bar{v}_2, ..., \bar{v}_k\}=sp\{S\}$ and its dimension (i.e., the number of linearly independent vectors in S) is denoted *dim{S}*. The symbol *I* denotes the identity matrix. The rank of a matrix A is denoted *rank(A)* and the determinant of matrix A is represented by *detA*. The set of rational numbers, the real space and the set of integers are denoted *Q*, *IR* and *Z*, respectively. The set of non-negative integers and the set of positive integers are denoted *N* and N^+, respectively. The empty set is denoted $\emptyset$ and the notation A−B denotes the set $\{x: x\in A, x\notin B\}$. The notation $|S|$ means the cardinality of set S and $|\alpha|$ represents the absolute value of scalar α. As a final remark, if the element a belongs to a set S, the notation $a \in S$ is used and this notation is "abused" to indicate also that a column vector $\bar{m}_j$ (or a row vector M_i) is a column (row) of a matrix M, i.e., $\bar{m}_j \in M$ ($M_i\in M$) means that $\bar{m}_j$ (M_i) is a column (row) of matrix M.

The algorithms of interest in this paper are the so-called uniform dependence algorithms defined as follows.

Definition 2.1 (Uniform dependence algorithm) A *uniform dependence algorithm* is an algorithm that can be described by an equation of the form

$$v(\bar{j})=f_{\bar{j}}\,(\,v\,(\,\bar{j}-\bar{d}_1\,),\,v\,(\,\bar{j}-\bar{d}_2\,),\,...,\,v\,(\,\bar{j}-\bar{d}_m\,)\,) \tag{2.1}$$

where

(1) $\bar{j}\in J\subset Z^n$ is an index point, J is the *index set* of the algorithm and $n\in N^+$ is the number of components of $\bar{j}$;

(2) $f_{\bar{j}}$ is the computation indexed by $\bar{j}$, i.e., a single-valued function computed "at point $\bar{j}$" in a single unit of time;

(3) $v(\bar{j})$ is the value computed "at $\bar{j}$", i.e. the result of computing the right hand side of (2.1) and

(4) $\bar{d}_i\in Z^n$, i=1, ..., m, $m\in N$ are *dependence vectors*, also called *dependencies*, which are constant (i.e. independent of $\bar{j}\in J$); the matrix $D=[\bar{d}_1, ..., \bar{d}_m]$ is called the *dependence matrix* and $rank(D) \leqq \min\{n, m\}$ is denoted by m′. □

The class of uniform dependence algorithms is a simple extension of the class of computations described by uniform recurrence equations [6]. The main difference is that uniform dependence algorithms allow for different functions to be computed (in a unit of time) at different points of the index set. >From a practical viewpoint, uniform dependence algorithms can be easily related to programs where (1) a single statement appears in the body of a multiply nested loop and (2) the indices of the variable in the left hand side of the statement differ by a constant from the corresponding indices in each reference to the same variable in the right hand side. Alternative computations can occur in each iteration as a result of a single conditional statement as long as data dependencies do not change. Nested loop programs with multiple statements can also use the techniques of this paper together with the alignment method discussed in [9] and [11]. For the purpose of this paper, only structural information of the algorithm, i.e., the index set J and the dependence matrix D, is needed. Other information such as what computations occur at different points and where and when input/output of variables takes place can be ignored. Therefore, a uniform dependence algorithm with index set J and dependence matrix D is hereon characterized simply by the pair (J, D). Also, as in Definition 2.1, the letters n, m and m′ always denote the dimension of points in J, the number of dependence vectors and the rank of the dependence matrix D, respectively.

Definition 2.2 (Algorithm dependence graph and connectivity) : The *dependence graph* of an algorithm (J, D) is the non-directed graph (J, E) where J is the set of nodes of the graph and $E=\{(\bar{j}', \bar{j}) : \bar{j} - \bar{j}'= \bar{d}_i \text{ or } \bar{j}'-\bar{j}=\bar{d}_i, \bar{d}_i\in D, \bar{j}', \bar{j}\in J\}$ is the set of edges. Two index points $\bar{j}$, $\bar{j}'$ are *connected* if there exist index points $\bar{j}_1, ..., \bar{j}_l\in J$ such that $(\bar{j}, \bar{j}_1), (\bar{j}_1, \bar{j}_2), ..., (\bar{j}_{l-1}, \bar{j}_l), (\bar{j}_l, \bar{j}')\in E$. □

Definition 2.3 (Independent partition, maximal independent partition and partitionability): Given an algorithm (J, D) and the corresponding dependence graph (J, E), let $P = \{J_1, ..., J_q\}$, $q\in N^+$, be a partition of J. If for any arbitrary points $\bar{j}_1\in J_i$ and $\bar{j}_2\in J_l$, $i\neq l$, $(\bar{j}_1, \bar{j}_2) \notin E$, then P is an *independent partition* of the algorithm (J, D). The sets J_i, i=1, ..., q, are called the *blocks* of partition P. For an independent partition P, if any two arbitrary points $\bar{j}$, $\bar{j}'\in J_i$, i=1, ..., q, are connected in the dependence graph, then P is the *maximal independent partition* of (J, D) and is denoted P_{max}. The cardinality of the maximal independent partition $|P_{max}|$ is referred to as the *partitionability* of the algorithm (J, D). □

Informally, an independent partition of the index set J is such that there are no dependencies between computations which belong to different blocks of the partition. In graph theoretical terms, each block of an independent partition of algorithm (J, D) corresponds to a component of its dependence graph (J, E).

Generally, the shape and the size of the index set influence the partitionability of the algorithm because of boundary conditions. Consider two algorithms (J, D) and (J′, D′) such that D′=D and $J'=J\cup\{\bar{j}\}$, i.e., they differ only in the size of the index sets. The corresponding dependence graphs (J, E) and (J′, E′) can be such that $\bar{j}_1$, $\bar{j}_2\in J$ are not connected in (J, E) but are connected in (J′, E′) because it is possible that $E'=E\cup\{(\bar{j}, \bar{j}_1), (\bar{j}, \bar{j}_2)\}$. In other words, $\bar{j}_1$ and $\bar{j}_2$ can belong to different blocks of the maximal independent partition of (J, D) but belong to the same block of the maximal independent partition of (J′, D′). Example 2.1 in [16] illustrates this concept.

The dependence of the partitionability of an algorithm (J, D) on the shape and size of its index set J is a complicated issue and has practical implications. For example, in many programs, the loop bounds are not known at compile time and partitions must be identified which are independent of the size and shape of the index set and based solely on data dependencies. To concentrate on the relationship between the structure of the dependence vectors and the partitionability of the algorithm, the following concepts are introduced.

Definition 2.4 (Pseudo-connectivity): Given an algorithm (J, D), two points $\bar{j}$, $\bar{j}'\in J$ are *pseudo-connected* if there exists a vector $\bar{\lambda}\in Z^m$ such that $\bar{j}=\bar{j}'+D\,\bar{\lambda}$. □

Definition 2.5 (Pseudo-independent partition, maximal pseudo-independent partition and pseudo-partitionability): Given an algorithm (J, D), let $P=\{J_1, ..., J_q\}$ be a partition of J. If any two arbitrary points $\bar{j}_1\in J_i$ and $\bar{j}_2\in J_l$, $i\neq l$, are not pseudo-connected, then P is a *pseudo-independent partition* of the algorithm (J, D). If P is a pseudo-independent partition and any two arbitrary points $\bar{j}$, $\bar{j}'\in J_i$, i=1, ..., q, are pseudo-connected, then P is the *maximal pseudo-independent partition* of (J, D) and is denoted P_{max}. The cardinality of the maximal pseudo-independent partition $|P_{max}|$ is referred to as the *pseudo-partitionability* of the the algorithm (J, D). □

In many practical cases, e.g., when "while" loops are present in a program, it is also convenient to consider algorithms whose index sets are arbitrarily large along one or more dimensions. The general case, i.e., when this applies to all dimensions, is captured in the following definition and is also considered in this paper.

Definition 2.6 (Semi-infinite index set): An index set J is *semi-infinite* if it takes the following form:

$$J = \{\bar{j} = [j_1, ..., j_n]^T : 0 \leqq j_i < \infty, i = 1, ..., n\} \tag{2.2}$$

□

Example 2.1 Consider the algorithm (J, D), where $D=\begin{bmatrix} 2 & -3 \\ -1 & 2 \end{bmatrix}$ and $J=N^2$ is semi-infinite, i.e., $J=\{\bar{j}=[j_1, j_2]^T: 0 \leqq j_1, \bar{j}_2 < \infty\}$. The index set J is partially shown in Figure 2.1. The maximal partition $P_{max}=\{J_1, J_2, J_3, J_4\}$ where $J_1=\{[0,0]^T\}$, $J_2=\{[1,0]^T\}$, $J_3=\{[0,1]^T, [2,0]^T\}$ and $J_4=\{\bar{j}: \bar{j} \in (J-\bigcup_{i=1}^{3} J_i)\}$. Points $\bar{j}_1=[0,0]^T$ and $\bar{j}_2=[0,1]^T$ are not actually connected in the dependence graph of the algorithm. However, they are pseudo-connected by Definition 2.4 because $\bar{j}_2=\bar{j}_1+D\bar{\lambda}$, $\bar{\lambda}=[3,2]^T$. Intuitively, $\bar{j}_1$ and $\bar{j}_2$ are connected through points $[2,-1]^T$, $[4,-2]^T$, $[6,-3]^T$ and $[3,-1]^T$ which are not in J. P_{max} is not a pseudo-independent partition. Because $\det D=1$, equation $D\bar{\lambda}=\bar{j}-\bar{j}'$ always has an integer solution for $\bar{\lambda}$. So any two arbitrary points in J are pseudo-connected to each other. This implies that there is only one pseudo-independent partition P={J} which is also the maximal pseudo-independent partition. □

At this point, some comments are in order. First, by Definitions 2.3 and 2.5, a pseudo-independent partition is also an independent partition regardless of the shape and size of the index set. However, an independent partition is not necessarily a pseudo-independent partition. This is due to the fact that two arbitrary $\bar{j}_1$, $\bar{j}_2 \in J$ are pseudo-connected if they are connected and the reverse is not necessarily true. Secondly, for practical purposes, it is sufficient and more efficient to identify pseudo-independent partitions instead of independent partitions for the reasons explained next. Blocks of independent partitions that are not blocks of a pseudo-independent partition and contain only a few index points (hereon called *boundary blocks*) always occur at or near the boundaries of an index set. This can be shown for the general case when J is semi-infinite. In fact, according to Lemma 3 in [6], there exists always a point $\bar{p}=[p_1, p_2, ..., p_n]^T \in J$ such that for any arbitrary points $\bar{j}=[j_1, j_2, ..., j_n]^T \in J$ and $\bar{j}'=[j'_1, j'_2, ..., j'_n]^T \in J$ beyond $\bar{p} \in J$ (i.e., $j_i \geqq p_i$ and $j'_i \geqq p_i$, 1=1, ..., n), $\bar{j}$ and $\bar{j}'$ are connected in the dependence graph if and only if they are pseudo-connected. Boundary blocks are typically such that their individual cardinalities are very small in relation to the sizes of the algorithm and pseudo-independent blocks. As a consequence, little additional speed-up can result from executing boundary blocks concurrently with other blocks. Moreover, assigning small boundary blocks and other large pseudo-independent blocks to different processors of a multiprocessor can cause a non-balanced load distribution and inefficient system operation. In addition, as pointed out before, when index sets are known only at run time, it is not possible to determine the boundary blocks. Finally, many algorithms are such that they have the same partitionability and pseudo-partitionability. For all of the above reasons, this paper considers hereon only the problem of identifying pseudo-independent partitions of an algorithm.

3. BASIC RESULTS

In this paper, independent algorithm partitions are determined by two types of vector, called partitioning vectors and separating vectors, which must satisfy certain conditions. Together with some auxiliary terminology they are introduced in Definitions 3.1 and 3.3. These definitions are followed by a theorem and an example which make clear the relation between these vectors and independent algorithm partitions.

Definition 3.1 (Partitioning vector, determining vector, equal partitioning vector and algorithm coefficient): Given an algorithm (J, D), $\Pi=[\pi_1, \pi_2, ..., \pi_n] \in Z^{1 \times n}$ is a *partitioning vector* of (J, D), if and only if it satisfies the following conditions.

(1) $\gcd(\pi_1, \pi_2, ..., \pi_n)=1$†.

(2) There exists a set of $m'=\text{rank}(D)$ linearly independent dependence vectors $\bar{d}_{t_1}, \bar{d}_{t_2}, ..., \bar{d}_{t_{m'}}$ such that

$$\Pi\bar{d}_{t_1}=...=\Pi\bar{d}_{t_{m'}}=disp\Pi>0. \tag{3.1}$$

The dependence vectors $\bar{d}_{t_1}, ..., \bar{d}_{t_{m'}}$ are called the *determining vectors* of Π. If $\Pi\bar{d}_i=0(\text{mod disp}\Pi)$††, i=1, ..., m, then Π is called an *equal partitioning vector* of (J, D). The constant $\alpha=\gcd(\text{disp}\Pi, \alpha_1, ..., \alpha_m)$ where $\alpha_i=\Pi\bar{d}_i\,(\text{mod disp}\Pi)$, i=1, ..., m, is called the *algorithm coefficient.* □

For a given partitioning vector the set of determining vectors is not necessarily unique and, therefore, dispΠ might not be unique, either. However, given a partitioning vector and a set of determining vectors, dispΠ is unique. Therefore, whenever dispΠ is mentioned, it is associated with a particular set of determining vectors.

By Definition 3.1, if m'=n, then for each set of determining vectors $\bar{d}_{t_1}, ..., \bar{d}_{t_{m'}}$, the corresponding partitioning vector Π is the unique solution that satisfies conditions 1 and 2 in Definition 3.1 and the following system of linear equations:

$$\begin{cases} \Pi(\bar{d}_{t_1}-\bar{d}_{t_2})=0 \\ \Pi(\bar{d}_{t_1}-\bar{d}_{t_3})=0 \\ ... \\ \Pi(\bar{d}_{t_1}-\bar{d}_{t_{m'}})=0 \end{cases} \tag{3.2}$$

When m'<n, the partitioning vector determined by m' linearly independent dependence vectors $\bar{d}_{t_1}, ..., \bar{d}_{t_{m'}}$ is not unique and of course, it belongs to the solution space of Equation 3.2. In the next section, a closed form expression is provided for a partitioning vector as a solution of Equation 3.2.

A partitioning vector Π defines a set of hyperplanes $\Pi\bar{j}=c(\text{mod}\alpha)$, $c \in Z$, in the index space. Because an index point lies on only one of the hyperplanes, the index set J can be partitioned according to them, i.e., all points $\bar{j}$ lying on hyperplanes such that, for a fixed c, $\Pi\bar{j}=c(\text{mod}\alpha)$, belong to the same block of the partition. The following definition states this concept formally.

Definition 3.2 (α-partition): Let Π be a partitioning vector and α be the algorithm coefficient for (J, D). The partition $P_\alpha=\{J_0, ..., J_{\alpha-1}\}$ where $J_i=\{\bar{j}: \bar{j} \in J, \Pi\bar{j}=i\,(\text{mod}\,\alpha)\}$, i=0, ..., α−1, is called the *α-partition* of (J, D). □

Clearly, P_α is a partition and it is shown in Theorem 3.1 that P_α is also a pseudo-independent partition.

For the case where m' < n, i.e., rank(D) < n, a necessary condition for two index points $\bar{j}_1, \bar{j}_2 \in J$ to be pseudo-connected is that equation $D\bar{x}=(\bar{j}_1-\bar{j}_2)$ has at least a real solution $\bar{x} \in \mathbb{R}^m$. This motivates the introduction of the following concepts. Let row vector Ψ_i be such that $\Psi_i D=\bar{0}$. Clearly, there are n−m' linearly independent such vectors, denoted $\Psi_1, ..., \Psi_{n-m'}$, and they define a set of

† : gcd $(a_1, ..., a_n)$=the greatest common divisor of $a_1, ..., a_n$.

†† : a=b (mod c) if and only if $a=b+a_1 c$, $a_1 \in Z$.

hyperplanes

$$\begin{bmatrix} \Psi_1 \\ \dots \\ \Psi_{n-m'} \end{bmatrix} \bar{j} = \bar{y}, \quad \bar{y} \in Z^{n-m'}, \tag{3.3}$$

in the index space. The index set J can be partitioned such that points lying on the same hyperplane belong to the same block of the partition. It will be clear later that if two index points $\bar{j}_1, \bar{j}_2 \in J$ lie on the same hyperplane defined by (3.3), the equation $D\bar{x} = (\bar{j}_1 - \bar{j}_2)$ has a solution. These concepts are formally defined as follows.

Definition 3.3 (Separating vector and separating matrix): Given an algorithm (J, D), $\Psi_i = [\psi_{i1}, \dots, \psi_{in}] \in Z^{1\times n}$ is a *separating vector* of (J, D) if and only if it satisfies the following conditions.

(1) $\gcd(\psi_{i1}, \dots, \psi_{in}) = 1$

(2) $\Psi_i D = \bar{0}$.

Let $\Psi_1, \dots, \Psi_{n-m'}$ be all the linearly independent separating vectors; the matrix $\Psi = \begin{bmatrix} \Psi_1 \\ \dots \\ \Psi_{n-m'} \end{bmatrix}$ is called *separating matrix.* □

A set of $n-m'$ linearly independent separating vectors $\Psi_1, \dots, \Psi_{n-m'}$ for algorithm (J, D) can be found by solving the equation in condition 2 of Definition 3.3. The following definition indicates how to use these separating vectors to construct a corresponding algorithm partition.

Definition 3.4 (Ψ-partition): Let Ψ be a separating matrix of algorithm (J, D). The partition $P_\Psi = \{J_{\bar{y}_1}, \dots, J_{\bar{y}_q}\}$ of J is called the *Ψ-partition* of algorithm (J, D) if $J_{\bar{y}_i} = \{\bar{j}: \bar{j} \in J, \Psi\bar{j} = \bar{y}_i\}$, where $\bar{y}_i = [y_{1i}, \dots, y_{(n-m')i}]^T \in Z^{(n-m')}$ is called the *index* of block $J_{\bar{y}_i}$, i=1, ..., q. □

Clearly, P_Ψ is a partition of J. If m'=n, then $P_\Psi = \{J\}$ is a trivial partition because the only separating vector is $\bar{0}$ in this case. As for P_α, P_Ψ is actually pseudo-independent as shown later in Theorem 3.1. Let $J_{\bar{y}} \in P_\Psi$ and consider the subalgorithm $(J_{\bar{y}}, D)$. Clearly, if $\alpha > 1$, subalgorithm $(J_{\bar{y}}, D)$ can be further partitioned by the partitioning vector Π. In other words, the index set J can be partitioned by a set of hyperplanes

$$\begin{bmatrix} \Pi \\ \Psi \end{bmatrix} \bar{j} = \begin{bmatrix} y_0 \,(\mathrm{mod}\,\alpha) \\ \bar{y} \end{bmatrix}, \; y_0 \in \{0, 1, \dots, \alpha-1\} \text{ and } \bar{y} \in Z^{n-m'}, \tag{3.4}$$

and all points lying on the same hyperplane belong to the same block of the partition. This partition is formally stated next.

Definition 3.5 (αΨ-partition): Let Π be a partitioning vector and Ψ be a separating matrix of algorithm (J, D). The partition $P_{\alpha\Psi} = \{J_{\bar{y}_1}, \dots, J_{\bar{y}_k}\}$ of index set J is called the *αΨ-partition* if $J_{\bar{y}_i} = \{\bar{j}: \bar{j} \in J, \begin{bmatrix} \Pi\bar{j}(\mathrm{mod}\ \alpha) \\ \Psi\bar{j} \end{bmatrix} = \bar{y}_i\}$, where $\bar{y}_i = [y_{0i}, y_{1i}, \dots, y_{(n-m')i}]^T \in Z^{n-m'+1}$ is called the *index* of block $J_{\bar{y}_i}$, i=1, ..., k. □

Partitioning vectors and separating vectors play a very important role in algorithm partition. The next theorem gives some of the motivation for the introduction of these concepts. More specifically, it provides sufficient conditions for two computations to belong to different blocks of an independent partition, in terms of those vectors and the index points associated with the computations. Moreover, it shows that α−partitions, Ψ−partitions and αΨ-partitions are all pseudo-independent.

Theorem 3.1 : Let Π be a partitioning vector, α be the algorithm coefficient and Ψ be a separating matrix of algorithm (J, D), respectively. The following statements are true:

(1) For any two arbitrary points $\bar{j}_1, \bar{j}_2 \in J$, if $\Pi\bar{j}_1 \neq \Pi\bar{j}_2 (\mathrm{mod}\ \alpha)$ then they are not pseudo-connected. Therefore, P_α is a pseudo-independent partition of (J, D).

(2) For any two arbitrary points $\bar{j}_1, \bar{j}_2 \in J$, if $\Psi\bar{j}_1 \neq \Psi\bar{j}_2$ then they are not pseudo-connected. Therefore, P_Ψ is a pseudo-independent partition of (J, D).

(3) $P_{\alpha\Psi}$ is a pseudo-independent partition. □

Proof: Provided in Appendix.

Corollary 3.1: If algorithm (J, D) has an equal partitioning vector Π, then $\bar{j}_1, \bar{j}_2 \in J$ are not pseudo-connected if $\Pi\bar{j}_1 \neq \Pi\bar{j}_2$ (mod dispΠ) or $\Psi\bar{j}_1 \neq \Psi\bar{j}_2$. □

As a particular case of Theorem 3.1, Corollary 3.1 is obviously true. If algorithm (J, D) has an equal partitioning vector Π, then the algorithm coefficient α=dispΠ. By Theorem 3.1, Corollary 3.1 holds.

Example 3.1: Consider algorithm (J, D) where $J = \{[j_1, j_2]^T: 0 \leqq j_1, j_2 \leqq s, s \in N^+\}$ and $D = [\bar{d}]$ where $\bar{d} = [2, 2]^T$. Figure 3.1 shows the index set J for s=4. There is only one possible set of determining vectors $\{\bar{d}\}$. One of the partitioning vectors determined by $\bar{d}$ is $\Pi = [-1, 2]$. It follows that $\mathrm{disp}\Pi = \Pi\bar{d} = 2$ and the algorithm coefficient α=2. Consider index points $\bar{j}_1 = [0, 0]^T$ and $\bar{j}_2 = [1, 0]^T$; because $\Pi\bar{j}_1 = 0(\mathrm{mod}\alpha)$ and $\Pi\bar{j}_2 = 1(\mathrm{mod}\alpha)$, by Theorem 3.1, they are not pseudo-connected. There is only one linearly independent separating vector $\Psi_1 = [1, -1]$ and a separating matrix is $\Psi = [1, -1]$. Again, consider index points $\bar{j}_1, \bar{j}_3 = [0, 1]^T$ for which $\Psi\bar{j}_1 = 0$ and $\Psi\bar{j}_3 = -1$. By Theorem 3.1, $\bar{j}_1$ and $\bar{j}_3$ are not pseudo-connected. In Figure 3.1 (a) and (b), hyperplanes $\Pi\bar{j} = c_1(\mathrm{mod}\alpha)$ and $\Psi\bar{j} = c_2$, $c_1, c_2 \in Z$, are drawn, respectively. All the points lying on the same hyperplane $\Pi\bar{j} = c_1\ (\mathrm{mod}\alpha)$ belong to the same block of the α-partition and all the points lying on the same hyperplane $\Psi\bar{j} = c_2$ belong to the same block of the Ψ-partition. Figure 3.1 shows the α−partition, Ψ-partition and αΨ-partition pictorially. Let s=3, then $P_\alpha = \{J_0, J_1\}$ where $J_0 = \{[0, 0]^T, [0, 1]^T, [0, 2]^T, [0, 3]^T, [2, 0]^T, [2, 1]^T, [2, 2]^T, [2, 3]^T\}$ and $J_1 = \{[1, 0]^T, [1, 1]^T, [1, 2]^T, [1, 3]^T, [3, 0]^T, [3, 1]^T, [3, 2]^T, [3, 3]^T\}$. Also $P_\Psi = \{J_{[-3]}, \dots, J_{[0]}, \dots, J_{[3]}\}$ where $J_{[3]} = \{[3, 0]^T\}$, $J_{[2]} = \{[2, 0]^T, [3, 1]^T\}$, $J_{[1]} = \{[1, 0]^T, [2, 1]^T, [3, 2]^T\}$, $J_{[0]} = \{[0, 0]^T, [1, 1]^T, [2, 2]^T, [3, 3]^T\}$, $J_{[-1]} = \{[0, 1]^T, [1, 2]^T, [2, 3]^T\}$, $J_{[-2]} = \{[0, 2]^T, [1, 3]^T\}$ and $J_{[-3]} = \{[0, 3]^T\}$. Interested readers can find $P_{\alpha\Psi}$ by intersecting $J_i \cap J_{[j]}$, i=0, 1 and j=−3, ..., 3, [16] which is such that $|P_{\alpha\Psi}| = 12 \leqq \alpha\,|P_\Psi| = 14$. Clearly, P_Ψ, P_α and $P_{\alpha\Psi}$ are pseudo-independent partitions. In Section 4, it is shown that the αΨ-partition is also the maximal pseudo-independent partition. □

By Theorem 3.1, if for any arbitrary value of $a \in Z$, $0 \leqq a < \alpha$ there is at least one point $\bar{j} \in J$ such that $\Pi\bar{j} = a(\mathrm{mod}\alpha)$, then $J_i \in P_\alpha$ is such that $J_i \neq \emptyset$, i=0, ..., α−1. Therefore, $|P_{\max}| \geqq \alpha$. Intuitively, if J is large enough and *dense* (informally, an index set J is dense if any arbitrary point $\bar{j} \in Z^n$ that is inside the boundaries of J belongs to J), then for any arbitrary value of a, $0 \leqq a < \alpha$ and $a \in Z$, there usually exists at least one index point $\bar{j}$ such that $\Pi\bar{j} = a(\mathrm{mod}\,\alpha)$. Therefore, it is reasonable to make the following assumption:

Assumption 3.1 (Index set): For an algorithm (J, D) under consideration in this paper, let Π be a partitioning vector and α be the algorithm coefficient. It is assumed that for any arbitrary value of $a \in Z$, $0 \leq a < \alpha$, there is at least one point $\bar{j} \in J$ such that $\Pi\bar{j} = a(\mathrm{mod}\alpha)$. □

In [16], it is shown that this is true if the index set J is defined by (2.2), i.e., $J=N^n$. Therefore, $|P_{max}| \geq \alpha$ if J is semi-infinite.

4. INDEPENDENT PARTITIONING METHOD

In this section, Theorem 3.1 and other results and concepts introduced in Section 3 are used to prescribe a partitioning procedure. Afterwards, Section 4.1 discusses how to find the partitioning vectors required by the procedure. Then Section 4.2 characterizes algorithms for which the method yields the optimal partition and derives lower and upper bounds on the pseudo-partitionability of arbitrary uniform dependence algorithms. The independent partitioning procedure is as follows:

Procedure 4.1 (Finding $\alpha\Psi$-partition for algorithm (J, D)):

Input: Algorithm (J, D).

Output: $\alpha\Psi$–partition $P_{\alpha\Psi}$ for algorithm (J, D).

Step 1: Select m' linearly independent dependence vectors $\bar{d}_{t_1}$, ..., $\bar{d}_{t_{m'}}$, set $D_c=[\bar{d}_{t_1}, ..., \bar{d}_{t_{m'}}]$, find $T\in Z^{m'\times n}$ such that $rank(TD_c)=m'$ and compute the corresponding partitioning vector Π according to Theorem 4.1 provided in Section 4.1. If $disp\Pi \neq |det(TD_c)|$, then select another set of m' linearly independent dependence vectors and compute the corresponding partitioning vector until all distinct sets of m' linearly independent dependence vectors are considered. If a partitioning vector Π such that $disp\Pi = |det(TD_c)|$ is not found, then select the partitioning vector Π such that $(|det(TD_c)|)/(disp\Pi)$ is minimum. Then compute the algorithm coefficient α according to Definition 3.1.

Step 2: Obtain $n-m'$ linearly independent separating vectors Ψ_1, ..., $\Psi_{n-m'}$ by solving equation $\Psi_i D=\bar{0}$. Set $\Psi=\begin{bmatrix}\Psi_1\\ \dots \\ \Psi_{n-m'}\end{bmatrix}$.

Step 3: For every index point $\bar{j}\in J$, if $\begin{bmatrix}\Pi\bar{j}(mod\alpha)\\ \Psi\bar{j}\end{bmatrix}=\bar{y}_i$ $=[y_{0i}, y_{1i}, ..., y_{(n-m')i}]^T$, then assign $\bar{j}$ to $J_{\bar{y}_i}$, the block indexed by $\bar{y}_i$, i.e., $\bar{j}\in J_{\bar{y}_i}$.

Step 4: $P_{\alpha\Psi}=\{J_{\bar{y}_1}, ..., J_{\bar{y}_k}\}$. □

4.1. Finding a partitioning vector

This subsection provides in Theorem 4.1 a closed form expression for the computation of partitioning vector Π, as required in Step 1 of Procedure 4.1. Given m' linearly independent vectors $\bar{d}_{t_1}$, ..., $\bar{d}_{t_{m'}}$, the corresponding partition vector Π belongs to the solution space of Equation 3.2. In [3], a closed form expression for a partitioning vector which is determined by $\bar{d}_{t_1}$, ..., $\bar{d}_{t_{m'}}$ is given. This result is restated as Theorem 4.1 as follows.

Theorem 4.1 [3] : Let $\bar{d}_{t_1}$, ..., $\bar{d}_{t_{m'}}$ be linearly independent, consider matrix $D_c=\left[\bar{d}_{t_1}, ..., \bar{d}_{t_{m'}}\right]$ and let $T\in Z^{m'\times n}$ be such that $rank(TD_c)=m'$. Then $\Pi=\beta\bar{1}(TD_c)^{-1}T$ is a partitioning vector determined by $\bar{d}_{t_1}$, ..., $\bar{d}_{t_{m'}}$ and $disp\Pi=\beta$, where $\beta\in N^+$ is such that $\Pi\in Z^{1\times n}$ and the greatest common divisor of the n components of Π is equal to one. □

Notice that matrix $T\in Z^{m'\times n}$ (such that $rank(TD_c)=m'$) always exists. Because $rank(D_c)=m'$, there are m' linearly independent rows in D_c. Suppose rows r_1, ..., $r_{m'}$ are linearly independent. If $T=\begin{bmatrix}E_{r_1}\\ \dots \\ E_{r_{m'}}\end{bmatrix}$, where E_{r_1}, ..., $E_{r_{m'}}$ are as defined in the beginning of Section 2, then $rank(TD_c)=m'$. In other words, the result of multiplying D_c by T is a square submatrix of D that contains exactly m' linearly independent rows of the m' linearly independent columns of D. If $m'=n$, then $T=I$, the identity matrix, and $\Pi=\beta\bar{1}D_c^{-1}$. The essence of the proof is as follows [3]. Because $\beta\bar{1}(TD_c)^{-1}TD_c=\beta\bar{1}$, vector $\beta\bar{1}(TD_c)^{-1}T$ satisfies Equation 3.1 and meets conditions 1 and 2 in Definition 3.1 by the meaning of the constant β; so $\Pi=\beta\bar{1}(TD_c)^{-1}T$ is a partitioning vector determined by $\bar{d}_{t_1}$, ..., $\bar{d}_{t_{m'}}$ and $disp\Pi=\beta>0$.

When considering partitioning vectors for a given algorithm, it is desirable to obtain an equal partitioning vector because of the simplicity and regularity of the resulting partitions. Necessary and sufficient conditions for an algorithm to have an equal partitioning vector are provided in [16].

4.2. Sufficient conditions for optimality

Theorem 3.1 provides a necessary condition for two index points in J to be pseudo-connected. Next it is shown in Theorems 4.2 and 4.3 that this condition becomes sufficient when the dependence matrix D satisfies certain constraints. The implication of this result is that the partition $P_{\alpha\Psi}$ obtained by Procedure 4.1 is maximal. In order to motivate and facilitate the understanding of the main results of this section, a special case is first discussed in Theorem 4.2 where $m'=n$, i.e, $rank(D)=n$. In this case, the Ψ–partition is trivial, i.e., $P_\Psi=\{J\}$.

Theorem 4.2 : Let $m'=n$, Π be a partitioning vector of algorithm (J, D) determined by $\bar{d}_{t_1}$, ..., $\bar{d}_{t_n}$, $D_c=[\bar{d}_{t_1}, ..., \bar{d}_{t_n}]$ and α be the algorithm coefficient. If $|detD_c|=disp\Pi$, then

(1) two index points $\bar{j}_1, \bar{j}_2\in J$ are pseudo-connected if and only if $\Pi\bar{j}_1=\Pi\bar{j}_2(mod\alpha)$;

(2) the α-partition is the maximal pseudo-independent partition of (J, D), i.e., $P_{max}=P_\alpha$, and $|P_{max}|=\alpha$. □

Proof: Provided in [16].

In this case, Procedure 4.1 becomes very simple. Because $rank(D)=n$, there is only one trivial separating vector $\bar{0}$ and therefore, Ψ–partition$=\{J\}$. So Step 3 in Procedure 4.1 can be skipped. When Π is an equal partitioning vector, then $\Pi\bar{d}_i=0(moddisp\Pi)$, i=1, ..., m. So $\alpha=disp\Pi=|detD_c|$. This fact is summarized as Corollary 4.1 as follows.

Corollary 4.1 : Let $m'=n$, Π be an equal partitioning vector of algorithm (J, D) determined by $\bar{d}_{t_1}$, ..., $\bar{d}_{t_n}$ and $D_c=[\bar{d}_{t_1}, ..., \bar{d}_{t_n}]$. If $|detD_c|=disp\Pi$, then the pseudo-partitionability of (J, D) is equal to the absolute value of the determinant of matrix D_c, i.e., $|P_{max}|=|detD_c|$. □

The meaning of Corollary 4.1 is as follows. For a class of algorithms, the number of blocks in the maximal pseudo-independent partition is equal to $|detD_c|$, the absolute value of the determinant of a submatrix of the dependence matrix D. If the algorithm is to be executed by clusters of processors with limited inter-cluster communication capabilities then the number of clusters to be used should be directly related and perhaps equal to the cardinality of the pseudo-independent partition. In such MIMD systems, $|detD_c|$ is a direct indication of how many clusters can be used to execute the algorithm. The next theorem discusses the sufficient condition of optimality for general cases.

Theorem 4.3 : Consider algorithm (J, D), let $\bar{d}_{t_1}$, ..., $\bar{d}_{t_{m'}}$ be linearly independent, $D_c=[\bar{d}_{t_1}, ..., \bar{d}_{t_{m'}}]$, $T\in Z^{m'\times n}$ be such that $rank(TD_c)=m'$, $\Pi=disp\Pi\bar{1}(TD_c)^{-1}T$ be the partitioning vector determined by $\bar{d}_{t_1}$, ..., $\bar{d}_{t_{m'}}$, α be the algorithm coefficient and Ψ be a separating matrix. If $|det(TD_c)|=disp\Pi$, then

(1) two points $\bar{j}_1$, $\bar{j}_2 \in J$ are pseudo-connected if and only if $\Pi(\bar{j}_1 - \bar{j}_2) = 0 \pmod{\alpha}$ and $\Psi\bar{j}_1 = \Psi\bar{j}_2$;

(2) the $\alpha\Psi$-partition is the maximal pseudo-independent partition of (J, D), i.e., $P_{\max} = P_{\alpha\Psi}$.

(3) $|P_\Psi| \leqq \prod_{i=1}^{n-m'} (x_i + 1)$, where $x_i = \max\{\Psi_i(\bar{j}_1 - \bar{j}_2)$: $\bar{j}_1, \bar{j}_2 \in J\}$, i=1, ..., n-m', and $\alpha \leqq |P_{\max}| \leqq \alpha\, |P_\Psi|$. □

Proof: Provided in [16].

If the cardinalities of the α-partitions of sub-algorithms ($J_{\bar{y}_i}$, D), where $J_{\bar{y}_i} \in P_\Psi$, i=1, ..., q, are all equal to α, then $|P_{\max}| = \alpha\, |P_\Psi|$. However, for some block $J_{\bar{y}} \in P_\Psi$, the cardinality of its α-partition might be less than α because for some value of $a \in Z$, $0 \leqq a < \alpha$, there might not exist an index point $\bar{j} \in J_{\bar{y}}$ such that $\Pi\bar{j} = a \pmod{\alpha}$. This phenomenon is illustrated in the following example.

Example 4.1: Consider the algorithm of Example 3.1 with s=3. There is only one set of determining vectors $\{\bar{d}\}$ and $D_c = D$. If $T = [-1, 2]$, then $TD_c = [2]$. According to Theorem 4.1, $\Pi = 2\,\bar{1}\,[2]^{-1}\,[-1, 2] = [-1, 2]$ and $\mathrm{disp}\Pi = 2 = \det(TD_c)$. As in Example 3.1, the separating matrix is $\Psi = [1, -1]$. To illustrate Theorem 4.3 (1), consider points $\bar{j}_1 = [0, 0]^T$ and $\bar{j}_2 = [2, 2]^T$. Because $\Pi\bar{j}_1 = \Pi\bar{j}_2 \pmod{\alpha}$ and $\Psi\bar{j}_1 = \Psi\bar{j}_2$, by Theorem 4.3, they are pseudo-connected. Due to the fact that $\mathrm{disp}\Pi = \det(TD_c)$, by Theorem 4.3, $P_{\alpha\Psi}$ is the maximal pseudo-independent partition. Consider $J_{[3]} \in P_\Psi$, i.e., the block whose points $\bar{j}$ are such that $\Psi\bar{j} = 3$. $J_{[3]} = \{[3, 0]^T\}$ as found in Example 3.1. There does not exist an index point $\bar{j} \in J_3$ such that $\Pi\bar{j} = 0 \pmod{\alpha}$. This illustrates the explanation before this example. By Theorem 4.3 (3), $|P_\Psi| = x + 1 = 7$, where $x = 3 - (-3) = 6$ and $|P_{\max}| = 12 \leqq \alpha\, |P_\Psi| = 14$. □

5. COMPARISON WITH MINIMUM DISTANCE METHOD

In the minimum distance method [11], [13], an elegant idea is used which consists of using a linear mapping to transform the dependence matrix D into an upper triangular matrix denoted D^t in [11]. These two dependence matrices are equivalent in the sense that each dependence vector in D^t, is a linear integer combination of the dependence vectors in D and vice versa. A set of initial points, each of which corresponds to a block in the resulting partition, is identified by D^t and the cardinality of the partition is the product of the diagonal elements of D^t. The original program is transformed into a parallel program containing parallel statements such as "parallel do" by D^t. An independent partition is implicitly expressed by D^t and a set of initial points.

In relation to the terminology used in this paper, a clarification needs to be made regarding the ability of the minimum distance method to find the maximal independent partition. In fact, as the next example illustrates, that method finds the maximal pseudo-independent partition instead of the maximal independent partition which is claimed by the authors of [11], [13].

Example 5.1: Consider the algorithm of Example 2.1. In Example 2.1, the maximal independent partition of this algorithm is obtained and it has four blocks, i.e., $|P_{\max}| = 4$. By the minimum distance method, the upper triangular matrix is $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. So, there is only one block in the partition obtained by the minimum distance method, which, clearly, is not maximal. However, it is the maximal pseudo-independent partition. □

Unfortunately, the minimum distance method finds the maximal pseudo-independent partition only for a restricted class of algorithms as illustrated in the next two examples. Two possible interpretations are considered for the following definition of $D^c_{m\times n}$ in line 15, page 218 of [11], "$D^c_{m\times n}$ (corresponding to D in this paper) contains only those linear-independent dependence cycles (corresponding dependence vectors in this paper)." In one interpretation, it is assumed that only $m' \leqq n$ linearly independent vectors are taken into account and included in $D^c_{m\times n}$ and the remaining vectors are ignored. In the other interpretation it is assumed that all dependence vectors are included in $D^c_{m\times n}$. The next two examples illustrate the fact that both interpretations result in incorrect results.

Example 5.2 Consider algorithm (J, D) where J is semi-infinite and $D = \begin{bmatrix} 3 & 0 & 0 \\ -3 & 2 & 3 \end{bmatrix} = [\bar{d}_1 \bar{d}_2 \bar{d}_3]$. By the method proposed in this paper, if $\bar{d}_1$, $\bar{d}_2$ are chosen as determining vectors, then $D_c = \begin{bmatrix} 3 & 0 \\ -3 & 2 \end{bmatrix}$, the corresponding partitioning vector $\Pi = [5, 3]$ and the algorithm coefficient $\alpha = \gcd(\Pi\bar{d}_1 (\mathrm{mod\,disp}\Pi),\ \Pi\bar{d}_2 (\mathrm{mod\,disp}\Pi),\ \Pi\bar{d}_3 (\mathrm{mod\,disp}\Pi)) = \gcd(0, 0, 3) = 3$. Because $\mathrm{disp}\Pi = \det D_c$, by Theorem 4.2, the α-partition (which is equal to the $\alpha\Psi$-partition) for this algorithm is pseudo-maximal and there are three blocks in the maximal pseudo-independent partition. There are two sets of two linearly independent dependence vectors $\{\bar{d}_1, \bar{d}_2\}$ and $\{\bar{d}_1, \bar{d}_3\}$. By the minimum distance method, if $\bar{d}_1$, $\bar{d}_2$ are included in D^c, i.e., $D^c = \begin{bmatrix} 3 & -3 \\ 0 & 2 \end{bmatrix}$ (note that D^c in [11] is D_c^T in this paper), then the corresponding upper triangular matrix is $\begin{bmatrix} 3 & -3 \\ 0 & 2 \end{bmatrix}$ and the number of blocks in the maximal pseudo-independent partition is 6. If $\bar{d}_1$, $\bar{d}_3$ are chosen to be in D^c, the corresponding upper triangular matrix is $\begin{bmatrix} 3 & -3 \\ 0 & 3 \end{bmatrix}$ and the number of blocks in the maximal pseudo-independent partition is 9. Recall that the number of blocks in the maximal pseudo-independent partition is three. Therefore, both cases yield partitions that are not independent. So all the dependence vectors have to be taken into account to find the maximal pseudo-independent partition instead of only m' linearly independent dependence vectors. □

Example 5.3 Consider an algorithm (J, D) with n dependence vectors and n−1 linearly independent dependence vectors, i.e., $D \in Z^{n\times n}$ and rank(D)=n−1. By the minimum distance method, if all dependence vectors are included in the dependence matrix, then $D^c = D^T \in Z^{n\times n}$. The upper triangular matrix D^t is square and $D^t = K \times D^c$ and all diagonal elements of D^t are positive. This implies that $\mathrm{rank}(D^t) = n$. However, because $\mathrm{rank}(D^c) = n-1$, $\mathrm{rank}(D^t) \leqq n-1$, this is a contradiction. □

In summary, the minimum distance method is valid only for the case where all dependence vectors are linearly independent. When m'=m=n, it generates the maximal pseudo-independent partition and when m'=m<n, it generates an independent partition that may not be maximal. In [13], an algorithm to generate initial points is presented for this case. However, its complexity and optimality are not clear. Moreover, only index sets $J = \{[j_1, ..., j_n]^T$: $0 \leq j_i \leqq s_i$, i=1, ..., n, $s_i \in N^+\}$ are considered.

In addition, compared with the partitioning method proposed in this paper, the minimum distance method has the following disadvantages. First, in the minimum distance method, partitions are expressed implicitly in terms of the upper triangular matrix and a set of initial points. According to [11], to find the upper triangular matrix, it is necessary to solve n integer programming problems with m variables which are NP-complete, where n, m are the number of dimensions of the index points and the number of dependence vectors, respectively. This is expensive although it is affordable when n, m are small. In the method proposed in this paper, partitions are expressed explicitly in terms of the partitioning vectors and separating vectors. To obtain these vectors, the dominating computations required are to find partitioning vectors, i.e., consider at most all possible combinations of m' vectors from the m dependence vectors and compute $\mathrm{disp}\Pi\bar{1}\,(TD_c)^{-1}T$. The complexity is bounded above by $\binom{m}{m'}O(n^3)$.

Secondly, as mentioned above, in the minimum distance method, blocks of the resulting partition are implicitly expressed in terms of the upper triangular matrix and a set of initial points. Although the serial loops in the original program can be transformed into parallel loops by the upper triangular matrix, it is costly to obtain the explicit expression of blocks of the partition and, especially, to know which block a given index point belongs to. According to the notations in [11], given an index point $X \in Z^{1\times n}$, one way to see which block it belongs to is to see if equation $X = X_{i0} + AD^t$ has an integer solution $A \in Z^{1\times n}$, where X_{i0} is an initial point belonging to block i. If it has, then X belongs to block i. If it does not, then another initial point X_{j0} belonging to block j, $j \neq i$, is tried until an initial point X_{k0} is found such that equation $X = X_{k0} + AD^t$ has an integer solution. This can be a very computationally expensive procedure. In contrast, in the method proposed in this paper, blocks of partitions are explicitly expressed in terms of the vectors. To see which block a given index point $\bar{j} \in Z^n$ belongs to, the computations required are to compute $\Pi\bar{j}(\mathrm{mod}\,\alpha)$ and $\Psi\bar{j}$.

In the method proposed in this paper, for some algorithms that do not satisfy the condition in Theorem 4.3, the resulting pseudo-independent partition may not be maximal. The problem of finding the maximal pseudo-independent partitions for these algorithms is the subject of current continuing research.

6. OPEN PROBLEMS AND CONCLUSIONS

Basically, there are two open problems. First, when algorithms do not satisfy the conditions described in Theorems 4.2 and 4.3, it is not known whether or not the independent partition obtained by the proposed method is maximal. Extensions of the approach proposed in this paper to deal with those cases are currently under investigation. Secondly, the upper bound provided in Theorem 4.3 is not tight. As discussed in the proof of Theorem 4.3 in [16], for some value of $a \in Z$, $0 \leqq a < \alpha$, there may not exist an index point $\bar{j} \in J_{\bar{y}} \in P_\Psi$ such that $\Pi\bar{j} = a\,(\mathrm{mod}\,\alpha)$ although it is assumed that there always exists at least one index point $\bar{j}' \in J$ such that $\Pi\bar{j}' = a\,(\mathrm{mod}\,\alpha)$. Let $a_1, \ldots, a_\beta \in Z$, $0 \leqq a_i < \alpha$, $i=1, \ldots, \beta$, be such that there exists at least one index point $\bar{j} \in J_{\bar{y}}$ and $\Pi\bar{j} = a_i(\mathrm{mod}\alpha)$, $i=1, \ldots, \beta$, then, obviously, a tighter upper bound is $|P_{max}| \leqq \beta\,|P_\Psi|$. To find the number β is still open.

The main contribution of this paper is a computationally inexpensive method for identifying independent partitions of algorithms with uniform dependencies. For a large class of algorithms the resulting partitions are maximal. The partitioning method proposed here can be applied in practice as one of the many analysis procedures used by optimizing compilers to detect and exploit concurrency in serial programs. It may be particularly useful in mapping algorithms into multiprocessor machines where processors are organized in clusters with limited inter-cluster communication capabilities. In these systems, different clusters can process distinct blocks of a partition without inter-cluster communication overhead costs. Among others, such multiprocessors include Cedar [17] and Cm* [5].

ACKNOWLEDGEMENTS

The authors thank the referees for their suggestions for improvements in the presentation of this paper.

REFERENCES

[1] U. Banerjee, S.C. Chen, D.J. Kuck and R.A. Towle, "Time and Parallel Processor Bounds for FORTRAN-like Loops", IEEE Trans. on Comp., Vol. c-28, No. 9, pp.660-670, Sept. 1979.

[2] R. Cytron, "Doacross: Beyond Vectorization for Multiprocessors," Proc. 1986 Int'l Conf. on Parallel Processing, 1986.

[3] J.A.B. Fortes, "Algorithm Transformations for Parallel Processing and VLSI Architecture Design" , Ph.D. Thesis, Dept of Elect. Eng.-Syst., University of Southern California, December 1983.

[4] D. D. Gajski and J.-K. Peir, "Essential Issues in Multiprocessor Systems," IEEE Computer, Vol.18, No.6, pp. 9-27, June 1985.

[5] K. Hwang and F.A. Briggs, "Computer Architecture and Parallel Processing," McGraw-Hill, New York, 1984.

[6] R.M. Karp, R.E. Miller and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," JACM 14, 3, pp. 563-590, Jul. 1967.

[7] D.C. Kuck, A.H. Sameh, R. Cytron, A.V. Veidenbaum, C.D. Polychronopoulos, G. Lee, T. McDaniel, B.R. Leasure, C. Beckman, J.R.B. Davies and C. Kruskal, "The Effects of Program Restructuring, Algorithm Changes, and Architecture Choice on Program Performance," Proc. 1984 Int'l Conf. on Parallel Processing, pp.129-138, Aug. 1984.

[8] D.I. Moldovan and J.A.B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," IEEE Trans. Computers, Vol. C-35, No. 1, pp. 1-12, Jan. 1986.

[9] D. A. Padua, "Multiprocessors: Discussion of Theoretical and Practical Problems," Ph.D Thesis, Univ. of Illinois at Urb.-Champ., Rept. No. UIUCDCS-R-79-990, Nov. 1979.

[10] D.A. Padua, D.J. Kuck and D.L. Lawrie, "High Speed Multiprocessor and Compilation Techniques," IEEE Trans. on Computers, Vol. C-29, No. 9, pp. 763-776, Sept. 1980.

[11] J.-K. Peir and R. Cytron, "Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors," Proc. 1987 Int'l Conf. on Parallel Processing, pp.217-225, 1987.

[12] J.-K. Peir and D.D. Gajski, "CAMP: A Programming Aid for Multiprocessors," Proc. 1986 Int'l Conf. on Parallel Processing, pp.475-482, Aug. 1986.

[13] J.-K. Peir, "Program Partitioning and Synchronization on Multiprocessor Systems," Ph. D Thesis, Report No. UIUCDCS-R-86-1259, Dept. of Computer Science, Univ. of Illinois at Urb.-Champ., Urbana, Illinois, Mar. 1986.

[14] C.D. Polychronopoulos, D.J. Kuck and D.A. Padua, "Execution of Parallel Loops on Parallel Processor Systems," Proc. 1986 Int'l Conf. on Parallel Processing , pp. 519-527,Aug. 1986.

[15] M.J. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois. Report No. UIUCDCS-R-82-1105, 1982

[16] W. Shang and J.A.B. Fortes, "Partitioning of Uniform Dependency Algorithms for Parallel Execution on MIMD/Systolic Systems," Technical Report 88-18, Purdue University, W. Lafayette, IN 47907, April 1988.

[17] D. Kuck, E. Davidson, D. Lawrie and A. Sameh, "Parallel Supercomputing Today and the Cedar Approach," Science, Vol. 231, pp. 967-974, Feb. 28, 1986.

APPENDIX

Proof of Theorem 3.1 :

(1). Suppose $\bar{j}_1$ and $\bar{j}_2$ are pseudo-connected, then there exists a vector $\bar{\lambda} = [\lambda_1, ..., \lambda_m]^T \in Z^m$ such that $\bar{j}_1 + D\,\bar{\lambda} = \bar{j}_2$. Therefore,

$$\Pi\bar{j}_1 + \Pi D\bar{\lambda} = \Pi\bar{j}_2 \quad \text{or} \quad \Pi\bar{j}_1 + \sum_{i=1}^{m} \lambda_i \Pi\bar{d}_i = \Pi\bar{j}_2$$

Let $\Pi\bar{d}_i = \alpha_i + a_i \text{disp}\Pi$, α_i, $a_i \in Z$, $0 \leqq \alpha_i < \text{disp}\Pi$, i=1, ..., m. So,

$$\Pi\bar{j}_2 - \Pi\bar{j}_1 = \left(\sum_{i=1}^{m} a_i\lambda_i\right)\text{disp}\Pi + \sum_{i=1}^{m} \alpha_i\lambda_i$$

where $\sum_{i=1}^{m} \lambda_i a_i$ is an integer because λ_i and a_i, i=1, ..., m, are integers. Since gcd(dispΠ, α_1, ..., α_m)= α, $\alpha_i = \alpha\,\gamma_i$ and $\text{disp}\Pi = \alpha\,\gamma$, γ_i, $\gamma \in Z$, i=1, ..., m. Then,

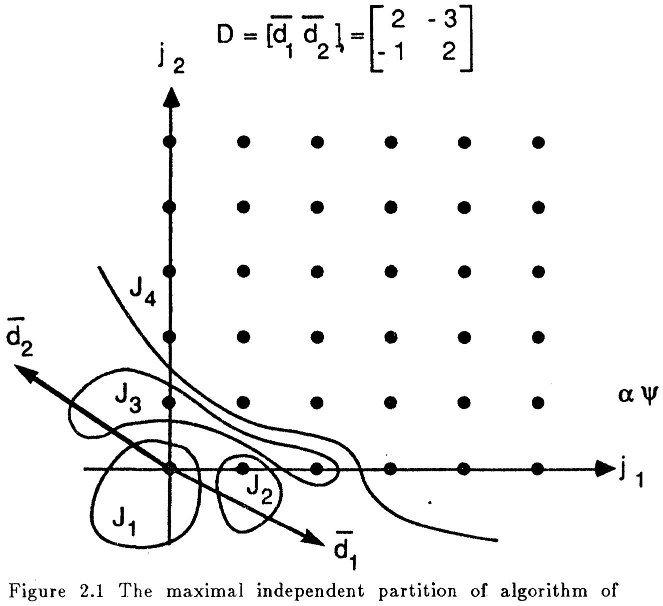
$$\Pi\bar{j}_2 - \Pi\bar{j}_1 = \alpha\left(\gamma \sum_{i=1}^{m} a_i\lambda_i + \sum_{i=1}^{m} \gamma_i\lambda_i\right) = 0 \pmod{\alpha}$$

i.e., $\Pi\,\bar{j}_1 = \Pi\,\bar{j}_2 \pmod{\alpha}$. This contradicts to the assumption. So $\bar{j}_1$ and $\bar{j}_2$ are not pseudo-connected.

Consider the α-partition P_α. Since $\Pi\bar{j} = i \pmod{\alpha}$, $\bar{j} \in J_i \in P_\alpha$, i=0, ..., $\alpha - 1$, for any two arbitrary index points $\bar{j} \in J_i$, $\bar{j}' \in J_l$, $i \neq l$, $\Pi\,\bar{j} \neq \Pi\bar{j}' \pmod{\alpha}$ and they are not pseudo-connected. By Definition 2.5, P is a pseudo-independent partition.

(2). Suppose that $\bar{j}_1$, $\bar{j}_2$ are pseudo-connected, then there exists a vector $\bar{\lambda} \in Z^m$ such that $D\bar{\lambda} = (\bar{j}_1 - \bar{j}_2)$. So, $\Psi D\bar{\lambda} = \Psi(\bar{j}_1 - \bar{j}_2)$. By Definition 3.3, $\Psi_i D = \bar{0}$, i=1, ..., n−m' which implies that $\Psi(\bar{j}_1 - \bar{j}_2) = \bar{0}$, i.e., $\Psi\bar{j}_1 = \Psi\bar{j}_2$. This is a contradiction to the assumption. So, $\bar{j}_1$, $\bar{j}_2$ are not pseudo-connected. For the Ψ-partition P_Ψ, let $\bar{j}_1 \in J_{\bar{y}_i}$ and $\bar{j}_2 \in J_{\bar{y}_l}$, $\bar{y}_i \neq \bar{y}_l$, $J_{\bar{y}_i}$, $J_{\bar{y}_l} \in P_\Psi$. The fact that $\bar{y}_i \neq \bar{y}_l$ implies that $\Psi\bar{j}_1 \neq \Psi\bar{j}_2$. So, $\bar{j}_1$, $\bar{j}_2$ are not pseudo-connected. By Definition 2.5, P_Ψ is pseudo-independent.

(3). Similarly, let $\bar{j}_1 \in J_{\bar{y}_i} \in P_{\alpha\Psi}$ and $\bar{j}_2 \in J_{\bar{y}_l} \in P_{\alpha\Psi}$, $\bar{y}_i \neq \bar{y}_l$, where $\bar{y}_i = [y_{0i}, y_{1i}, ..., y_{(n-m')i}]^T$ and $\bar{y}_l = [y_{0l}, y_{1l}, ..., y_{(n-m')l}]^T$. Since $\bar{y}_i \neq \bar{y}_l$, there exists at least one dimension $t \in \{0, 1, ..., n-m'\}$ such that $y_{ti} \neq y_{tl}$. If t=0, then $\Pi\bar{j}_1 \neq \Pi\bar{j}_2 \pmod{\alpha}$ and by (1) of Theorem 3.1, $\bar{j}_1$ and $\bar{j}_2$ are not pseudo-connected. If $1 \leqq t \leqq n-m'$, then $\Psi_t\bar{j}_1 \neq \Psi_t\bar{j}_2$ and by (2) of Theorem 3.1, $\bar{j}_1$ and $\bar{j}_2$ are not pseudo-connected. So, by Definition 2.5, $P_{\alpha\Psi}$ is pseudo-independent. □



$D = [\bar{d}_1\ \bar{d}_2] = \begin{bmatrix} 2 & -3 \\ -1 & 2 \end{bmatrix}$
j_2
j_1
J_1
J_2
J_3
J_4
$\bar{d}_1$
$\bar{d}_2$

Figure 2.1 The maximal independent partition of algorithm of Example 2.1 is $P_{max} = \{J_1, J_2, J_3, J_4\}$. However, there is only one block in the maximal pseudo-independent partition. Pictorially, only the connectivities of points near boundaries of J are influenced.

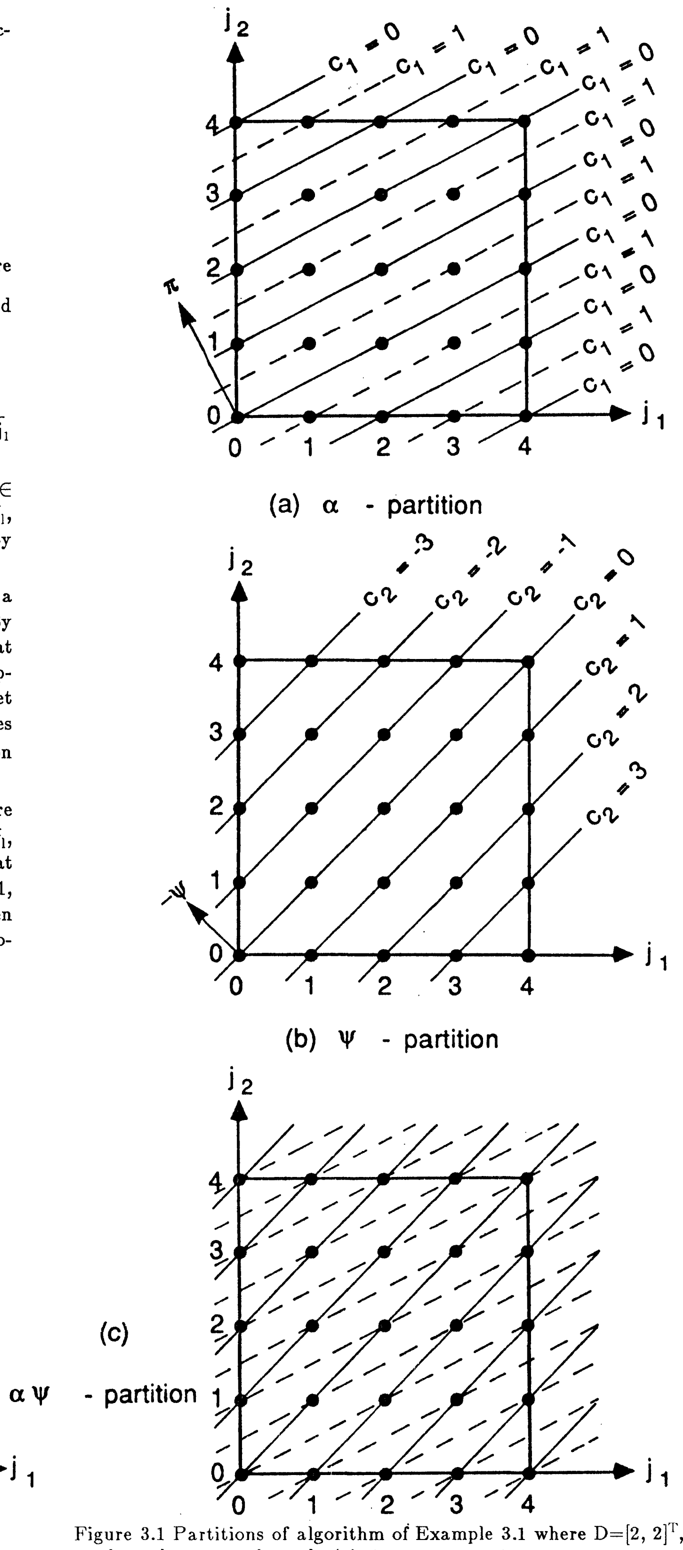


Figure 3.1 Partitions of algorithm of Example 3.1 where D=[2, 2]^T, Π=[−1, 2] and Ψ=[1, −1]. (a) α-partition: the hyperplanes are described by $\Pi\bar{j} = c_1 (\text{mod} 2)$. Points lying on dotted lines belong to $J_0 \in P_\alpha$ and points lying on solid lines belong to $J_1 \in P_\alpha$. (b) Ψ-partition: the hyperplanes are described by $\Psi\bar{j} = c_2$. Points lying on hyperplane $\Psi\bar{j} = c_2$ belong to $J_{[c_2]} \in P_\Psi$. (c) $\alpha\Psi$-partition: dotted lines specify α-partition and solid lines specify Ψ-partition.

Automatic Compound Function Definition for Multiprocessors[†]

Harlan E. Husmann[‡]
Supercomputer Systems, Inc.
1414 W. Hamilton Avenue
Eau Claire, WI 54701

David J. Kuck
Center for Supercomputing
Research and Development
University of Illinois
Urbana, IL 61801

David A. Padua
Center for Supercomputing
Research and Development
University of Illinois
Urbana, IL 61801

Abstract

We compare two algorithms for automatically defining compound functions (tasks) derived from a single program for a multiprocessor architecture. Both algorithms assume a compound function is represented by a Fortran DO loop. The first algorithm bases its decision on whether a particular loop defines a compound function on the amount of loop parallelism that is available in the loop. The loop's parallelism must be more than a threshold value that is supplied as a parameter to the algorithm. The second algorithm uses more information than the first algorithm when picking compound functions. It estimates a loop's execution time and it compares this estimate with the estimates for other loops, then it picks compound functions such that the program's execution time estimate is minimized. Using the Parafrase system, we apply these compound function definitions to 61 Fortran programs and compare the speed-up each compound function definition yields for each program.

1. Introduction

Two factors provide the primary motivation for this study. First, many vendors of high-performance computers are moving toward tightly-coupled multiprocessor architectures. Some examples are Cray Research (Cray X-MP, Cray-2, Cray Y-MP, Cray-3), ETA Systems (ETA-10), and Alliant Computer Systems (FX/8). Second, a large body (tens of millions of lines) of Fortran programs exist for high-performance computers. Users would like to execute these codes efficiently on new computers (and, thus, new architectures) as they become available, without a large amount of reprogramming.

The above two factors lead us to investigate restructuring serial Fortran programs for a multiprocessor architecture. In particular we are interested in this study to see how well a compiler can automatically partition programs into tasks for parallel processing on a hypothetical high-performance multiprocessor, and we compare two compiler algorithms for doing this.

For this study, a program is represented by a directed graph called a **program graph**, with the nodes of the graph representing computation and the arcs of the graph representing the execution ordering between nodes. We call a node of the graph a **CF (compound function)** of which there are two types: **CTF's (control functions)** which represent serial computation and **CPF's (computation functions)** which represent parallel computation. We concentrate our study on CPF's.

Both of our algorithms for finding CF's assume that a Fortran DO loop marks the beginning and the end of a CPF. The first algorithm bases its decision about whether a loop is a CPF on the amount of parallelism that is available in the loop. The loop's parallelism must be more than a threshold value for the loop to be chosen as a CPF. We call this CF definition algorithm the **Loop Parallelism CF Definition.**[1]

[†]This work was supported in part by the National Science Foundation under Grant Nos. US NSF DCR84-10110 and US NSF DCR84-06916, the U. S. Department of Energy under Grant No. US DOE-DE-FG02-85ER25001, the IBM Donation, and the Alliant Computer Systems Corporation.

[‡]The author was a research assistant at the Center for Supercomputing Research and Development for this work.

[1]Section 3 explains the CF Definition algorithms in more detail. We use the phrase "CF definition" to mean the process of picking or defining which loops of a program are CPF's.

The second algorithm incorporates more information into deciding which loops become CPF's: it estimates a loop's execution time on our multiprocessor model and compares this estimate with the estimates for other loops. The algorithm then picks the set of CPF's that minimize the program's execution time estimate. We call this CF definition algorithm the **Optimal CF Definition.**

We use the Parafrase system to empirically evaluate and compare these two CF definition algorithms. Using Parafrase, we apply both CF definitions to, and estimate execution time for, 61 Fortran programs.

Section 2 discusses preliminary material including our multiprocessor model, program execution on our multiprocessor, the Parafrase system, and the measurements we make with Parafrase. In Section 3 we explain the two CF definition algorithms. Section 4 presents the data we gathered with Parafrase on the 61 Fortran programs.

2. Background Material

2.1. Our Serial Computer and Multiprocessor Models

Figure 1 shows our multiprocessor model. The multiprocessor has many arithmetic processing elements called **PE's**. Each PE possesses a **local memory** that only it can access. All PE's can access a shared memory called **global memory**. An intermediate memory (in terms of access time and accessibility) called **cluster memory** exists between the fully-shared global memory and the private local memory. PE's can share data with other PE's through global memory or through cluster memory but not through another PE's local memory. A group of PE's, called a **cluster**, share cluster memory. For this study we assume eight PE's form a cluster and that there are four clusters in our multiprocessor.

Data transfer between the PE's and the global memory is done via an **interconnection network**, such as an Omega network [9]. We assume that the network has a constant **round-trip time** (the time it takes a message to travel from a PE through the network to the global memory and back to the PE) regardless of the load on the network. The access time for global memory through the interconnection network is slower (e.g., factor of three to ten) than the access time of local or cluster memory.

Programs access two types of data through the network: **items** and **blocks**. An item is either a scalar or an array element and it takes the full round-trip time to access (read or write) the global memory. A block is a number of elements from a data array. We call the maximum number of elements in a block the **block size** and it is small in size (e.g., 32) with constant stride. We abbreviate block size with **BS**. Programs can store or fetch blocks, which we call a **block store** and a **block fetch**, respectively; **block access** refers to either a block store or a block fetch. We assume block access is only available in CPF's.

The network can pipeline block accesses in a manner similar to the operation of pipelined arithmetic units in vector computers like the Cray-1 [19]. For example, the first element of a block fetch takes the full round-trip time to reach the PE, but each of the remaining elements takes a fraction of the round-trip time.

The other model we use is a **serial computer**. The serial computer is similar to one PE from the multiprocessor with only scalar instructions (each of the multiprocessor's PE's can execute scalar instructions or vector instructions up to length BS). We assume that the serial computer has enough registers and pipelining to make all fetches free—only stores contribute to a program's execution time on the serial computer. In the multiprocessor, we count stores *and* fetches, which differs from previous Parafrase–based studies [2] [3] [7]. There are two reasons why we choose to count fetches. First, although for a serial computer one can argue that we need not count fetches because the time spent fetching data is masked by pipelining, adequate memory bandwidth, CPU registers, caches, etc., we feel we cannot make such an argument for our multiprocessor model because global memory speed is much slower than the speed of the PE's. The other reason we count fetches is that we are interested in the amount of program execution time that is due to memory access (on the multiprocessor) and we want to measure this as accurately as possible.

The scalar arithmetic operation speed of the serial computer exactly matches the scalar arithmetic operation speed of a PE, and the memory speed of the serial computer exactly matches the local memory speed of a PE. We use the serial computer as the basis for calculating the speed–up of executing a program on our multiprocessor.

The multiprocessor architecture we use in this study takes advantage of local memory, cluster memory, and the vector–like block access of the interconnection network. We call this architecture the **Block Access–No Overlap Architecture** (the **A3** multiprocessor for short), following the naming scheme in [6]. At compile time, Parafrase allocates as much program data as possible to A3's local and cluster memory. Parafrase also transforms the program for block access for this multiprocessor. As the name implies, the A3 multiprocessor can perform block accesses but cannot *overlap* block accesses: when the program has to perform a block access, it suspends until the block access is complete.

We denote the speed–up of a program executing on the A3 multiprocessor with S_p^n, where p is the number of PE's in the multiprocessor and n is the architecture number. We assume the A3 multiprocessor has 32 PE's (four clusters of eight PE's each) throughout this work. Thus, we denote A3's speed–up with S_{32}^3.

We calculate speed–up for a program executing on the A3 multiprocessor in the following manner. We estimate the program's execution time on the serial computer; we call this estimate T_1. We estimate the program's execution time on the A3 multiprocessor; we call this estimate T_{32}^3. The program's speed–up is the ratio of T_1 and T_{32}^3.

$$S_{32}^3 = \frac{T_1}{T_{32}^3}$$

Parafrase uses the original program, before transforming it for block access, to calculate T_1.

We denote a program's speed–up using the Optimal CF Definition for the A3 multiprocessor with

$$S_{32}^3(\mathrm{Opt})$$

and similarly for the Loop Parallelism CF Definition.

$$S_{32}^3(\mathrm{LP})$$

We compare these two speed–ups with a ratio denoted with

$$R(\mathrm{LP/Opt}) = \frac{S_{32}^3(\mathrm{LP})}{S_{32}^3(\mathrm{Opt})}.$$

2.2. The Loops We Study

The Fortran programs we use in this study are serial programs originally written for serial computers. We investigate automatically transforming these serial programs for parallel execution on the A3 multiprocessor. We use the Parafrase system to transform a program for parallel execution [3] [7] [13] [14] [21]. Parafrase uses a data dependence graph to identify those parts of the program that can execute in parallel. The most common parallelism that Parafrase identifies is parallel execution of different loop iterations—this is what we study here. Parafrase has been used to study other types of parallelism [1] [20].

Parafrase identifies three different loop types for multiprocessors: serial loops, DoAll loops, and DoAcross loops. The iterations of a **serial loop** must execute in sequence (i.e., iteration i must finish before iteration i+1 can begin).

In terms of parallelism, **DoAll loops** represent the exact opposite of serial loops. All iterations of a DoAll loop can execute concurrently. If a DoAll loop has N iterations and N PE's are available, each PE can execute one iteration and the entire DoAll loop can complete in the time of one iteration. Padua discusses DoAll loops in [13] as do Lundstrom and Barnes in [12] and Davies in [4].

The third type of loop, the **DoAcross loop**, lies between a serial loop and a DoAll loop in terms of parallelism. Inter–iteration dependences (control and data) determine how much parallelism is available in a DoAcross loop.

Delay characterizes a DoAcross loop and it is the amount of time we expect it takes to satisfy an inter–iteration dependence[2] [3]. Each iteration of a DoAcross loop must "delay" an amount of time equal to the time needed to satisfy these dependences. In this context, DoAll loops can be thought of as DoAcross loops with a delay of zero and serial loops can be thought of as DoAcross loops with a delay equal to the execution time of the loop's body.

The execution time of a DoAcross loop is given by

$$\left\lceil \frac{N}{p} - 1 \right\rceil - 1 * \max(b, p*d) + d*((N-1) \bmod p) + b$$

where N is the loop bound, p is the number of PE's executing the loop, b is the execution time estimate of the loop body, and d is the DoAcross delay [15].

We use a percentage, $\mathbf{P_X}$, to characterize DoAcross loops: a larger $\mathbf{P_X}$ means a more parallel loop. A DoAll loop is a DoAcross loop with P_X=100% and a serial loop is a DoAcross loop with P_X=0%. P_X for a loop, given b and p, is calculated by the following formula.

$$P_X = (1 - \frac{d}{b}) * 100\%$$

An extensive discussion of the DoAcross loop is found in [3] and additional results are found in [16] and [17].

2.3. Program Execution on the Multiprocessor

As stated earlier, for execution on a multiprocessor a Fortran program is in the form of a directed graph called the program graph, with CF's containing computation from the serial Fortran program and the arcs representing the execution ordering of the CF's. A CF may represent one or more statements from the original program: a BAS (block of assignment statements), a loop and its body (including all loops in the body), a GOTO, an IF statement, etc. [8].

Recall that there are two types of CF's: CTF's and CPF's. CTF's represent the parts of the program that contain little or no parallelism, like control statements, and execute serially on one PE. CPF's represent the parts of the program with much computation and highly parallel parts—DoAll loops and DoAcross loops with a large P_X, for example—and execute on one or more clusters.

We concentrate our efforts on defining CPF's for our multiprocessor and we largely ignore CTF's. We only consider CTF's when we estimate a program's execution time and we always consider them serial computation. The automatic CF definitions we present here focus on defining CPF's such that we exploit as much loop parallelism as possible on our multiprocessor. After the CF definitions pick the CPF's, the CTF's are the parts of the program not represented by any CPF (i.e., the leftovers).

We restrict the definition of a CPF to a single loop because we assume only one level of parallelism is exploitable on our multiprocessor (i.e., only one loop at a time can execute its iterations in parallel on multiple PE's); we call such a loop a **CPF loop**. Because we have only one level of loop parallelism, a CPF loop "freezes" all

[2]Here we do not differentiate between control and data dependences.

loops surrounding it and all loops in its body. The CPF loop spreads its iterations onto multiple PE's; any loop inside a CPF loop executes serially on one PE and any loop enclosing a CPF loop executes serially as CTF's.

We allow only one level of parallelism because we believe this is a realistic assumption about the software and hardware that can be built today to support loop iterations executing in parallel. The Alliant FX/8 multiprocessor architecture is an example of such a system. It has software and hardware support for one level of loop parallelism within a computation complex [5].

We schedule loop iterations from a CPF loop across PE's at compile time such that iteration i of a CPF loop is scheduled to execute on PE ((i-1 mod *p*)+1), that is, iteration 1 executes on PE 1, iteration 2 on PE 2, and so on. Once we've assigned a group of *p* iterations, we must "fold back" loop iterations onto the PE's by scheduling iteration *p*+1 on PE 1, *p*+2 on PE 2, and so forth. All statements in the body of iteration i execute serially on PE ((i-1 mod *p*)+1). We call this type of loop scheduling **PE Prescheduling** because it schedules CPF loop iterations onto PE's before run time.

This work only considers parallel processing a single subroutine at a time, which means the spreading of different loop iterations from a single loop across multiple PE's. It does not consider parallel execution of subroutines, parallel execution of BAS's, or parallel execution of two *different* loops as in [20].

2.4. The Source Program Restructurer

As mentioned, we used the Parafrase system for this study. Parafrase performs source–to–source transformation of serial Fortran programs for parallel execution, using a data dependence graph to preserve the program's original semantics.

Parafrase accepts serial Fortran–66 programs and performs two sets of optimizations on them. The first set of optimizations are traditional compiler optimizations like induction variable removal, common subexpression elimination, invariant code floating, etc. The transformed source program then passes on to a set of architecture–specific optimizations. We target the second set of optimizations for a multiprocessor. These optimizations include recurrence recognition, loop fusion [20], DoAll and DoAcross loop recognition [3] [4], and several new transformations: CF definition, memory allocation, and block access generation [6]. The program is transformed for execution on our multiprocessor after the second set of optimizations finish. We only report on the CF definition transformations in this work.

3. Automatic Compound Function Definition

3.1. The Trade–off: Loop Parallelism vs. Loop Execution Time

The following program skeleton illustrates the trade-off we must deal with when defining CF's.

```
DoAcross 90% i=1,N
     S1
     . . .
     SM
     DoAll j=1,N
          SM+1
          . . .
          SN
     EndDoAll
     SN+1
     . . .
     SO
EndDoAcross
```

Since parallelism is limited to one level, only one loop in a multi–loop loop nest can define a CPF loop. There are two choices for a CPF loop in this program: the DoAcross loop or the DoAll loop. Which choice yields the most speed–up depends on the relative size (in terms of estimated execution time) of their loop bodies. If the two loops are perfectly nested, we can achieve more speed–up if the DoAll defines the CPF because it can use more PE's than the DoAcross loop (assuming N is large). If the DoAll loop is small compared to the DoAcross loop, however, we can speed up the program more if the DoAcross loop defines the CPF.

3.2. Loop Structure Representation

We view the structure of the loops in a Fortran program as a tree, as Figure 2 illustrates. Each loop from the program in Figure 2 corresponds to a node in the tree. Subtrees in the tree represent the nesting of loops in the program. The root node represents the program; all other nodes represent loops from the program. The root node's children, the level–one nodes, represent the program's outermost loops (loops 1, 2, and 9). Any loop within another loop is represented as a child node of the enclosing loop's node. This tree representation of a program's loop structure we call the **loop tree.**[3]

We consider two different approaches to defining CF's using the loop tree. The Loop Parallelism CF Definition is based on examining P_X for each node in the loop tree, and the Optimal CF Definition is based on a bottom–up traversal of the loop tree, estimating the execution time of each node it traverses. The Optimal CF Definition tries to use more information than just P_X to find CPF's. When evaluating a particular node as a CPF candidate, the Optimal CF Definition considers the node's execution time estimate on the A3 multiprocessor model, and the execution time estimates for other nodes. After comparing the execution time estimates, the Optimal CF definition picks CPF's that yield the largest speed–up (i.e., the minimum execution time estimate) for the entire loop tree.

3.3. The Different Ways A Loop Can Execute On Our Multiprocessor

Before presenting the two CF definitions let us show the four ways a node can execute on our multiprocessor. We illustrate this in Figure 3 from the point of view of node 2. The left–hand side of the figure shows a program outline (showing only loop boundaries) and node numbers. The right–hand side shows a column for each way node 2 can execute: it can execute as CTF's with its body also executing as CTF's (column 1); it can execute as CTF's with descendent nodes (e.g., node 3) executing as CPF's (column 2); it can execute as a CPF (column 3); and it can execute serially with an ancestor node (e.g., node 1) as a CPF (column 4). These are the only four ways that node 2 can execute on our multiprocessor, due to the restriction of one level of loop parallelism.

3.4. Loop Parallelism CF Definition

The Loop Parallelism CF definition algorithm uses a node parallelism threshold value, **T**, and chooses the nodes at the highest level of the loop tree with $P_X \geq T$. This is done with a breadth–first search of the loop tree, examining each branch of the tree and stopping the search along any branch where a node is found with $P_X \geq T$. The first node found down a branch with a $P_X \geq T$ is chosen as a CPF. If a node has $P_X < T$, we bypass the node and consider its children as possible CPF's. For example, if we use T=100% and apply the Loop Parallelism CF Definition to the loop tree in Figure 2, it will choose loop 6 as the only CPF; if we use T=90%, it will choose loops 4, 6, and 10 as CPF's; and if we use T=50%, it will choose loops 1, 3, 6, 8, 10, and 11 as CPF's.

3.4.1. The Heuristic for the Loop Parallelism CF Definition

One can see in Figure 2 that for T=100%, the Loop Parallelism CF Definition misses nodes that are good candidates for CPF's (e.g., nodes 4, 5, 10, and 11, all with $P_X \geq 80\%$). To try to exploit these nodes, we add a heuristic to the Loop Parallelism CF Definition. After the Loop Parallelism CF Definition picks CPF's with sufficient parallelism (i.e., $P_X \geq T$), the heuristic examines each *unfrozen* node that was *not picked by the Loop Parallelism CF Definition*, looking for nodes at the highest level in the loop tree with $P_X > 0\%$ (i.e., at least

[3]Note that the terms "node" and "loop tree" can be interchanged with the terms "loop" and "program," respectively. For the remainder of this section we use "node" and "loop tree" in our discussion.

some parallelism). We refer to the set of nodes picked by the Loop Parallelism CF Definition as the **original CPF's**, and any additional nodes picked by the heuristic as **additional CPF's**. From here on when we refer to the Loop Parallelism CF Definition we are referring to the algorithm with the heuristic.

Space limitations prevent us from giving a full algorithmic description of the Loop Parallelism CF Definition; one is given in [6]. Figure 4 shows which nodes would be picked as original CPF's and as additional CPF's from the loop tree of Figure 2 for different values of T. For T=100%, nodes 2 and 7 are not chosen as additional CPF's because they are frozen by node 6 being an original CPF. Node 3 is chosen over nodes 4 and 5 because it is at a higher level in the loop tree and it satisfies the heuristic ($P_X > 0\%$).

We do not go into an analysis of the running time of the Loop Parallelism CF Definition, but simply say that it does O(L) execution time estimates[4]; an analysis is done in [6]. The algorithm is basically two breadth–first searchs of the loop tree, the first picking original CPF's and the second picking additional CPF's. The advantages of this algorithm are its simplicity and that it need only do O(L) execution time estimates. The disadvantage of the Loop Parallelism CF Definition is it may not pick CPF's in the best way. For example, if we use $T \leq 90\%$ on the skeleton program from Section 3.1, we always choose the DoAcross loop as the CPF when choosing the DoAll loop instead could result in more speed-up.

3.5. The Optimal CF Definition

The Optimal CF Definition algorithm, shown in Figure 5, calculates for each node an execution time estimate for each of the different ways the node can execute on the A3 multiprocessor (recall Figure 3). If the minimum estimate is the one assuming the node defines a CPF, we call the node a **CPF candidate**. After all estimates have been calculated for all nodes, the Optimal CF Definition picks CPF's such that the execution time estimate for the entire loop tree is minimized. Since the algorithm considers every possible way CPF's could be picked from the loop tree, we are assured that the set of CPF's which minimizes the execution time estimate for the loop tree is found.

The procedure *Estimate_Node_Time* in Figure 5 calculates the execution time estimates. After all execution time estimates are completed, the procedure *Optimal_CF_Definition* searchs the loop tree breadth–first for the outermost nodes marked as CPF candidates, stopping the search along any branch when a CPF candidate is found.

Procedure *Estimate_Node_Time* calculates execution time estimates for three of the four ways a node can execute. The estimates which are missing are the ones assuming a node's ancestor defines a CPF (column 4 of Figure 3). Memory allocation and transformation for block access can be different for a node, depending on which ancestor node defines a CPF and, thus, these estimates must be recalculated for a node when the algorithm estimates execution time for each of its ancestor nodes. This is illustrated with the following example program.

```
      Do i = 1, N
       Do j = 1, N
        Do k = 1, N
1         A(i,j,k) = A(i,j,k) + 1.0
2         If (j.GT.1) B(i,j,k) = A(i,j-1,k) / B(i,j,k)
        EndDo
       EndDo
      EndDo
```

For this example, assume 1 time unit for each local memory access, 2 time units per global memory access, and 1 time unit for an arithmetic operation; also assume array B is always a global array. If the i–loop is the CPF loop, array A can be allocated to local memory following the algorithm in [6]. The total time for statement 1 is then three: one for the addition and two for the local memory accesses of array A. The total time for statement two is seven: one for the comparison, two for the global memory accesses of array B, one for the divide, and one for the local memory access of array A. The total serial time for the k–loop is 10N. If, however, the j–loop is the CPF loop, array A cannot be allocated to local memory and is allocated to global memory. Loop k's time now rises to 13N because of the change in memory allocation, which was necessary because a different loop surrounding the k–loop is the CPF loop.

[4]L is the number of nodes in the loop tree.

When *time_node_parallel* in line 7 of Figure 5 calculates an execution time estimate for a node assuming it executes as a CPF, it must redo memory allocation and block access transformation for each of the node's descendants. This forces a recalculation of execution time for each descendant, which results in an algorithm which does $O(L^2)$ execution time estimates in the worst case [6].

The advantage of the Optimal CF Definition is that it produces the set of CPF's that minimize the execution time estimates for the loop tree. The Optimal CF Definition finds the best CF definition because it calculates and compares execution time estimates for every possible way CPF's could be picked from the loop tree. The disadvantage is its $O(L^2)$ worst–case running time.

Figure 6 shows hypothetical execution time estimates and the data structures at various points while applying the Optimal CF Definition algorithm to the loop tree in Figure 2.

[18] takes the same approach as the Optimal CF Definition for allocating processors for a single level of parallelism. He has an O(L) algorithm, however, because he does not consider memory allocation or block access, and thus, a node's serial execution time estimate is the same no matter which ancestor node is a CPF node. We would also have a linear algorithm if we did not consider memory allocation and block access for nodes when estimating the execution time of ancestor nodes.

4. Automatic Compound Function Definition Experiments

4.1. The Programs We Use In Our Experiments

For the experiments we report here, we use three sets of programs: a Linpack kernel, an Eispack kernel, and a set of programs from ten benchmarks which we call the "Benchmark programs." The two kernels consists of all ***unique*** algorithms from their respective packages. For example, if there is a program in the Eispack package for real numbers, and an identical program for complex numbers, we keep only one of these programs for Eispack's kernel.

We change the kernels in the following ways to improve their performance and to allow Parafrase to analyze them: we remove the routine SGBFA from the Linpack kernel because it becomes too large after transforming it for block access, and we direct Parafrase to expand all CALL's to BLAS [10] in the Linpack kernel. This leaves 11 Linpack subroutines in the Linpack kernel. In the Eispack kernel we substitute the routine MUSEC1 for the routines BISECT and IMTQL2, as is done in [7]. There are 15 programs in the Eispack kernel.

The Benchmark programs are a set of 35 programs from several University of Illinois benchmarks which solve the ten different problems that appear below.

2–D Helmholtz	Linear Least Squares
3–D Helmholtz	Monte Carlo
Banded Factorization	Singular Value Decomposition
Conformal Mapping	Symmetric Linear System
Eigenvalue Problem	Symmetric Triangular Eigenvalues/vectors

4.2. How We Present the CF Comparison Data

We use R(LP/Opt) to compare the Loop Parallelism CF Definition and the Optimal CF Definition. Since the Optimal CF Definition is optimal for the A3 multiprocessor, a program's R(LP/Opt) is always between 0.0 and 1.0. A ratio of 1.0 means the Loop Parallelism CF Definition picks CPF's as well as the Optimal CF Definition; a ratio of 0.5 means the Loop Parallelism CF Definition picks CPF's in such a way that the program executes half as fast on the A3 multiprocessor as the CPF's the Optimal CF Definition picks.

We represent the three sets of programs with the histograms that appear in Figure 7, Figure 8, and Figure 9. Each histogram plots R(LP/Opt) for all programs in the set. We use T=100% for the Loop Parallelism CF Definition because we believe that if P_X is the only basis on which to pick CPF loops, we want to pick loops that have as much parallelism as possible.

4.3. Discussion of the CF Comparison Data

Looking at Figure 7 we see that 8 of the 11 Linpack programs show no difference between the two CF definitions. The 3 programs with R(LP/Opt) < 1.0 gain less than a factor of two in speed–up from the Optimal CF Definition: 1 program gains about 50% and 2 programs gain about 10%. This is not true of the Eispack programs, however. Figure 8 shows that nearly two–thirds of the programs benefit from the Optimal CF Definition, some by a large amount. Only 6 of 15 programs show no difference between the two CF definitions; 7 programs gain between 5% and 50%, 1 program gains nearly a factor of two, and another program gains over a factor of three from the Optimal CF Definition. Eispack is much more sensitive than Linpack to the way CF's are defined.

The Benchmark programs have their R(LP/Opt) plotted in Figure 9. Like Linpack, most of the programs show no improvement from the Optimal CF Definition; 26 of the 35 programs have a R(LP/Opt) of 1.0. But like Eispack, several programs gain significantly from the Optimal CF Definition. Six programs lie between 60% and 100%, 2 programs gain about a factor of two, and 1 program gains a factor of five from the Optimal CF Definition.

4.4. Reasons for R(LP/Opt) < 1.0

Figure 10 isolates and catalogs the programs with R(LP/Opt) < 1.0. There are four reasons for this: block access in serial loops, small DoAll loops in DoAcross loops, small loop bounds, and small DoAcross loops in DoAcross loops.

The most common reason for R(LP/Opt) < 1.0 is block accessing global arrays in serial loops. Block access is only available in CPF's and the heuristic of the Loop Parallelism CF Definition always passes over a loop with P_X=0% as a possible CPF, and thus, the loop never uses block access because it always executes as a CTF. Usually this is the correct decision. But in several cases a serial loop executes faster if Parafrase picks it as a CPF and transforms it for block access. Even though a serial loop cannot exploit any loop parallelism, it benefits from block accessing global arrays as a CPF instead of accessing their elements as single items as a CTF.

In Section 3 we presented small DoAll loops in DoAcross loops as an example of a drawback to the Loop Parallelism CF Definition. This is the situation where a small DoAll loop is in the body of a DoAcross loop with a large P_X. The two CF definitions could pick different loops to define CPF's. The Loop Parallelism CF Definition picks the DoAll loop (since P_X=100% for these experiments) as the CPF; the Optimal CF Definition examines both loops (the DoAcross loop as well as the DoAll loop) as possible CPF's. If the DoAcross loop is sufficiently parallel and sufficiently larger than the DoAll loop, the DoAcross loop executes faster if it defines the CPF and the DoAll loop executes serially, than if the DoAcross loop executes serially as a CTF and the DoAll loop defines the CPF. But with T=100% the Loop Parallelism CF Definition always picks the DoAll loop as the CPF. This is the second most common reason why programs have R(LP/Opt) < 1.0.

The third reason why the Optimal CF Definition performs better than the Loop Parallelism CF definition is small loop bounds, especially in the case of a pair of perfectly nested DoAll's. The Loop Parallelism CF Definition picks the outer DoAll as a CPF, regardless of the loop's upperbound. But if the outer DoAll has a very small upperbound compared to the inner DoAll, more parallelism is exploited if the inner DoAll is a CPF and the outer DoAll is a CTF. The Benchmark program THREEDH has this situation in three places.

The final reason for R(LP/Opt) < 1.0 is similar to the case of a small DoAll loop in a highly parallel DoAcross loop. Here, instead of a DoAll loop, we have a DoAcross loop inside another DoAcross loop. The heuristic of the Loop Parallelism CF Definition picks the outermost DoAcross loop if its $P_X > 0\%$. This may be the wrong decision if the inner DoAcross loop has a much higher P_X and their loop body sizes are not significantly different. As with the case of the DoAll in the DoAcross situation, the Optimal CF Definition does better than the Loop Parallelism CF Definition because it examines each loop as possible CPF's.

The table in Figure 10 shows the name of each program, the set of programs each program is from, and the reason why each program has R(LP/Opt) < 1.0. The number indicates how many distinct occurrences in each program (e.g., SPOCO has a small DoAll loop inside a DoAcross loop in two distinct places).

4.5. Conclusions

We see that the Optimal CF Definition improves the performance of a significant number of programs over the Loop Parallelism CF Definition. For Eispack, 60% of the programs show a performance improvement due to using the Optimal CF Definition; for all three sets of programs, 21 of 61 programs show an improvement by using the Optimal CF Definition over the Loop Parallelism CF Definition. The CF definition impacts S_{32}^{3} greatly for several programs. For example, one Benchmark program shows nearly a factor of *five* improvement in S_{32}^{3} by using the Optimal CF Definition.

Whereas 60% of Eispack's programs show an improvement in S_{32}^{3} by using the Optimal CF Definition, only 27% and 26% of the Linpack and Benchmark programs, respectively, show an improvement. The largest improvement for any Linpack program is about 50% while both Eispack and the Benchmarks have programs with improvement better than a factor of three. This illustrates that the relative importance of the Optimal CF Definition over the Loop Parallelism CF Definition depends on the workload. For some programs (e.g., Linpack) the Optimal CF Definition may not be critically important; however, other sets of programs (e.g., Eispack) suffer much more by using an inferior CF definition.

These results show why it is important to pick CPF's carefully. We could have a workload that is fairly independent of the CF definition, but if this is not the case, we lose much performance by using a less effective CF definition. And if maximizing program performance for all programs is critical, it is very important to use the best CF definition we can develop. In each set the S_{32}^{3} of several programs improve by using the Optimal CF Definition.

The most common reasons that the Optimal CF Definition produces better speed–ups are that it finds serial loops that benefit from block access and it picks a DoAcross loop as a CPF before an inner DoAll loop. In the first case, the Optimal CF Definition chooses a serial loop as a CPF when the loop can benefit from block accessing global variables (19 places in 13 programs). The Loop Parallelism CF Definition never considers a serial loop as a CPF because it never looks at loops with P_X=0%. Thus, all serial loops execute as CTF's. The second reason the Optimal CF Definition outperforms the Loop Parallelism CF definition is that it picks more parallel inner loops as CPF's before less parallel outer ones.

References

[1] Walid Abu–Sufah, H. E. Husmann, and D. J. Kuck, "On Input/Output Speedup in Tightly Coupled Multiprocessors," *IEEE Transactions on Computers*, Vol. C–35, No. 6, June 1986, pp. 520–530.

[2] Utpal Banerjee, S–C Chen, D. J. Kuck, and R. A. Towle, "Time and Parallel Processor Bounds for Fortran–Like Loops," *IEEE Transactions on Computers*, Vol. C–28, No. 9, September 1979, pp. 660–670.

[3] Ronald Gary Cytron, "Compile–time Scheduling and Optimization for Asynchronous Machines," Ph.D. Thesis, University of Illinois at Urbana–Champaign, DCS Report No. UIUCDCS–R–84–1177, October 1984.

[4] James Russell Beckman Davies, "Parallel Loop Constructs for Multiprocessors," Masters Thesis, University of Illinois at

Urbana–Champaign, DCS Report No. UIUCDCS–R–81–1070, May 1981.

[5] "A New Way to Speed Up a Supercomputer," *Electronics*, July 29, 1985, pp. 56–58.

[6] Harlan E. Husmann, "Compiler Memory Management and Compound Function Definition for Multiprocessors," Ph.D. thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana–Champaign, CSRD Report Number 575, August 1986.

[7] David J. Kuck, A. Sameh, R. Cytron, A. Veidenbaum, C. Polychronopoulos, G. Lee, T. McDaniel, B. Leasure, C. Beckman, J. Davies, and C. Kruskal, "The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance," *Proceedings of the 1984 International Conference on Parallel Processing*, August 1984.

[8] David J. Kuck, D. Lawrie, A. Sameh, and D. Gajski, "The Architecture and Programming of the Cedar System," *Proceedings of the 1983 LASL Workshop on Vector and Parallel Processing* , Los Alamos, NM, August 1983.

[9] Duncan H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Transactions on Computers*, Vol. C–24, No. 12, December 1975.

[10] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage", *ACM Transactions on Mathematical Software*, Vol. 5, No. 3, 1979.

[11] Kyungsook Lee, W. Abu–Sufah, and D. Kuck, "On Modeling Performance Degradation Due to Data Movement in Vector Machines," *Proceedings of the 1984 International Conference on Parallel Processing*, August 1984.

[12] Stephen F. Lundstrom and George H. Barnes, "A Controllable MIMD Architecture," *Proceedings of the 1980 International Conference on Parallel Processing*, 1980.

[13] David Alejandro Padua Haiek, "Multiprocessors: Discussion of Some Theoretical and Practical Problems," Ph.D. Thesis, University of Illinois at Urbana–Champaign, DCS Report No. UIUCDCS–R–79–990, November 1979.

[14] David A. Padua, D. Kuck, and D. Lawrie, "High–Speed Multiprocessors and Compilations Techniques," *IEEE Transactions on Computers,* Vol. C–24, No. 9, pp. 763–776, September 1980.

[15] Constantine D. Polychronopoulos and Utpal Banerjee, "Speedup Bounds and Processor Allocation for Parallel Programs on Multiprocessors," *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986.

[16] Constantine D. Polychronopoulos and Utpal Banerjee, "Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds," *IEEE Transactions on Computers*, Vol. C–36, No. 4, April 1987, pp. 410–420.

[17] Constantine D. Polychronopoulos, D. Kuck, and D. Padua, "Execution of Parallel Loops on Parallel Processor Systems," *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986.

[18] Constantine D. Polychronopoulos, "On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems," Ph.D. thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana–Champaign, CSRD Report Number 595, August 1986.

[19] Richard M. Russell, "The Cray–1 Computer System," *Communications of the ACM*, January 1978.

[20] Alexander Veidenbaum, "Compiler Optimizations and Architecture Design Issues for Multiprocessors," Ph.D. Thesis, University of Illinois at Urbana–Champaign, DCS Report No. UIUCDCS–R–85–1207, May 1985.

[21] Michael J. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. Thesis, University of Illinois at Urbana–Champaign, DCS Report No. UIUCDCS–R–82–1105, October 1982.

Figures

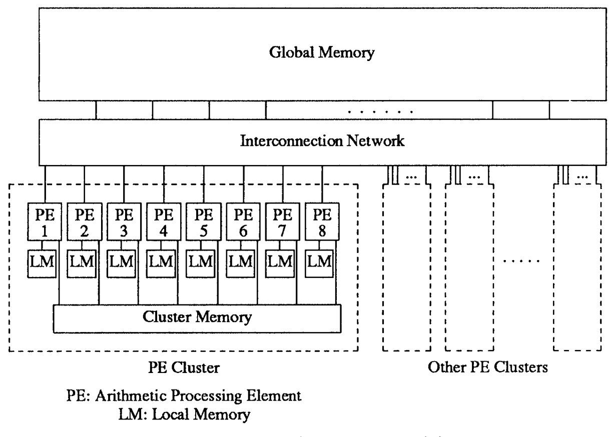


Figure 1. Multiprocessor Model

```
Program ''P''                    Loop Number
DoAcross 50%                          1
  . . .
EndDoAcross
DoAcross 5%                           2
  . . .
   DoAcross 50%                       3
      DoAcross 90%                    4
        . . .
      EndDoAcross
        . . .
      DoAcross 80%                    5
        . . .
      EndDoAcross
   EndDoAcross
   DoAll                              6
      . . .
      DoAcross 50%                    7
        . . .
      EndDoAcross
      . . .
   EndDoAll
   DoAcross 50%                       8
     . . .
   EndDoAcross
EndDoAcross
DoAcross 0%                           9
   DoAcross 90%                      10
     . . .
   EndDoAcross
   DoAcross 80%                      11
     . . .
   EndDoAcross
   . . .
EndDoAcross
```

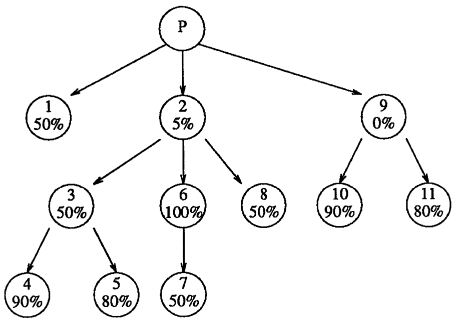


Figure 2. Program "P" and its Loop Tree Representation

Program Skeleton	Node Number	Different CF Definitions			
		1	2	3	4
DoAcross 50%	1	CTF	CTF	CFT	* CPF *
DoAcross 75%	2	CTF	CTF	* CPF *	(inside a CPF)
DoAcross 95%	3	CTF	* CPF *	(inside a CPF)	(inside a CPF)
. . .					
EndDoAcross					
EndDoAcross					
EndDoAcross					

Figure 3. The Different Ways Node 2 Can Execute

T	Original CPF's	Additional CPF's	Final Set of CPF's
100%	6	1,3,8,10,11	1,3,6,8,10,11
90%	4,6,10	1,5,8,11	1,4,5,6,8,10,11
50%	1,3,6,8,10,11	(none)	1,3,6,8,10,11

Figure 4. CPF's Pick by the Loop Parallelism CF Definition

```
    PROCEDURE Estimate_Node_Time(Node: node_ptr);
1.    inner_L := Node.child; /* Estimate leaf nodes first */
2.    DO WHILE (inner_L <> NIL)
3.        CALL Estimate_Node_Time(inner_L);
4.        inner_L := inner_L.next;
5.    OD;
6.    Node.T1  := time_node_serial(Node); /* Estimate time all different ways */
7.    Node.T_me_cpf := time_node_parallel(Node);
8.    IF (Node ∈ set_of_leaf_nodes) THEN
9.       Node.T_descendant_cpf := ∞ /* No descendants, so no estimate */
10.   ELSE
11.      Node.T_descendant_cpf := time_node_descendants_parallel(Node);
12.   FI;
13.   IF (T_me_cpf_minimum(Node)) THEN
14.       Node.cpf := TRUE; /* CPF candidate if parallel time is minimum */
15.   FI;
    END Estimate_Node_Time;

    PROCEDURE Optimal_CF_Definition;
      VAR Q: QUEUE; CPFS: SET;
16.   Q := empty();
17.   CPFS := {} ;
18.   DO Node := 1 TO #_level_one_nodes; /* Estimates time for all nodes */
19.       CALL Estimate_Node_Time(level_one_node(Node));
20.       CALL enqueue(Q,Node);
21.   OD;
22.   DO WHILE (empty(Q) == FALSE) /* Find biggest CPFS possible */
23.       Node := dequeue(Q);  /* Begin breadth-first search */
24.       IF (Node.cpf == TRUE) THEN
25.           CPFS := CPFS ∪ Node;
26.       ELSE /* Node rejected as CPF, examine its children */
27.           Node2 := Node.child;
28.           DO WHILE (Node2 <> NIL)  /* Add children to Q */
29.               CALL enqueue(Q,Node2);
30.               Node2 := Node2.next;
31.           OD;
32.        FI;
33.    OD;
    END Optimal_CF_Definition;
```

Figure 5. Optimal CF Definition

Node No.	Px	Execution Time Estimates			Node.cpf
		T1	T_me_cpf	T_descendant_cpf	
1	50	10	20	∞	
2	5	400	300	200	
3	50	70	60	50	
4	90	20	5	∞	TRUE
5	80	30	10	∞	TRUE
6	100	200	80	90	TRUE
7	50	70	30	∞	TRUE
8	50	20	30	∞	
9	0	150	50	40	
10	90	20	5	∞	TRUE
11	80	30	10	∞	TRUE

After Line Number:	Q	CPFS	Comment
21	1,2,9	∅	Execution time estimates finished.
32	2,9	∅	Node 1's T_me_cpf not minimal.
32	9,3,6,8	∅	Node 2's T_me_cpf not minimal.
32	3,6,8,10,11	∅	Node 9's T_me_cpf not minimal.
32	6,8,10,11,4,5	∅	Node 3's T_me_cpf not minimal.
32	8,10,11,4,5	{ 6 }	Node 6's T_me_cpf is minimal.
32	10,11,4,5	{ 6 }	Node 8's T_me_cpf is not minimal.
32	11,4,5	{ 6,10 }	Node 10's T_me_cpf is minimal.
32	4,5	{ 6,10,11 }	Node 11's T_me_cpf is minimal.
32	5	{ 4,6,10,11 }	Node 4's T_me_cpf is minimal.
33	(empty)	{ 4,5,6,10,11 }	Node 5's T_me_cpf is minimal.

Figure 6. Optimal CF Definition Time Estimates and Data Structures

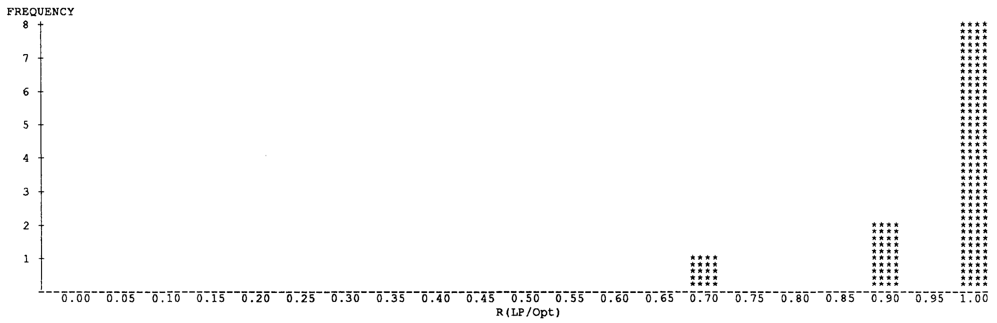


Figure 7. Linpack Speed-up Comparison

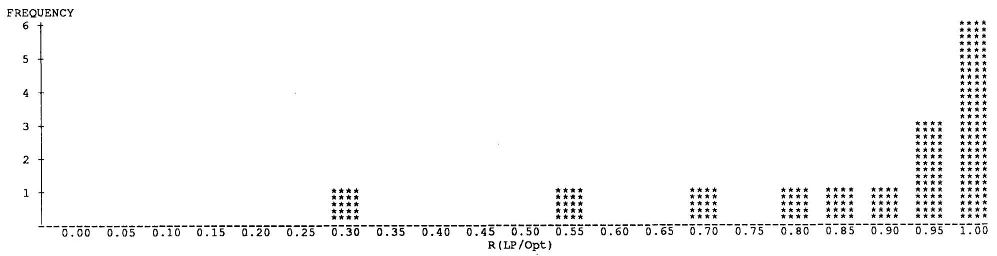


Figure 8. Eispack Speed-up Comparison

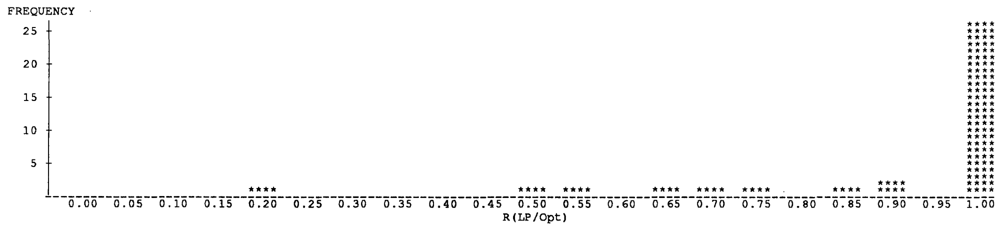


Figure 9. Benchmarks Speed-up Comparison

Program	Package	Block Access in Serial Loop	DoAll in DoAcross	Loop Bounds	DoAcross in DoAcross
SPBFA	L	3		1	
SPOCO	L		2		
SQRSL	L		1		
ELMBAK	E		1		
ELTRAN	E		1		
HQR	E	2			
HQR2	E	1			
IMTQLV	E				1
INVIT	E	2			
MINFIT	E	1	4		
MUSEC1	E	3			
TRED2	E		1		
COMPV0	B		1		
GENBUN	B	1			
ICSEVU	B	1			
RFFTF1	B	1			
SINT	B	1			
SORT3	B	1			
THREEDH	B			3	
TRID	B	1			
TRIX	B	1			

Figure 10. Reasons for R(LP/Opt) Less Than 1.0

Automatic Restructuring of Conditional Cyclic Loops

Gyungho Lee

The Center for Advanced Computer Studies
University of Southwestern Louisiana
P.O. Box 44330
Lafayette, LA. 70504

Abstract
This paper considers automatic restructuring of loops with conditional branching, especially a class of loops termed "conditional cyclic loops", for parallel processing. With a binary tree representation of a loop, parallelizing the loop on a shared memory machine allowing concurrent reads is discussed. In general, parallel execution of the loop consists of two stages, precomputation stage and path selection stage. The precomputation stage is equivalent to solving a set of recurrences, and the path selection stage is equivalent to solving a full-order Boolean recurrence. A few important special cases, which include "postfix-IF loops" and some of "linear mixed recurrence loops", can be executed in O(log n) time with a polynomial number of processors, where n is the loop bound.

1. Introduction

A loop with conditional branching is a typical dynamic control structure in ordinary sequential programs and is a major obstacle to automatic program restructuring for parallel processing. In general the loops with conditional branching can be classified by the availability of the values of the predicates defining the branching. Branching adds little difficulty to parallelizing a loop, provided that the value of the predicate at each iteration of the loop does not depend on the results of the previous iterations. This class of loops (*conditional acyclic loops*) is relatively easy to handle by using control or mode bits on synchronous array machines, or by using independent multiple control units on asynchronous multiprocessor machines. If branching is based on the values of the variables that are set as the results of the previous iterations of the loop, then the loop is difficult to parallelize. *Conditional cyclic loop* is a class of loops in which the conditional creates a dependence cycle across the loop index values, i.e. the values of the predicates are decided based on the branching taken in previous iterations. No attempts have been made to parallelize conditional cyclic loops except a simple special case. However, conditional cyclic loops are not rare in practice and seem to be a major unparallelized loop type in automatically restructured nonnumerical programs (see [7]). This paper concerns how to parallelize conditional cyclic loops in order to obtain a better speedup gain from automatically restructured programs.

Part of this work was done while the author was with Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign. This work was supported in part by the National Science Foundation under Grant No. US NSF DCR84-10110, and the U.S. Department of Energy under Grant No. US DOE DE-FG02-85ER25001.

Conditional cyclic loops can be classified further by the availability of the possible values of the variables that define the predicate of the IF statement. A *postfix-IF loop* is a conditional cyclic loop where the two possible values of the variables (one for the true branch and the other for the false branch) at each iteration of the loop do not depend on the previous iterations [11]. All the possible values are immediately available in a postfix-IF loop, and executing the loop will choose a particular value at each iteration for each variable. A *mixed recurrence loop* is a conditional cyclic loop where the variables used in branching cause recurrences. So, even the possible values, not alone the actual value at each iteration, of the variables are not available until the recurrences are solved. The recurrences can be linear or nonlinear. We consider the case of linear recurrence only (*linear mixed recurrence loop*). The overhead involved in parallelizing nonlinear mixed recurrence loops seems to overwhelm potential benefit of parallelizing them, and the cases of nonlinear recurrences are rare [7]. See Figure 1 for the examples of loops with conditional branching.

For the convenience of presentation, log n will denote $\log_2 n$ and will be assumed to have an integer value in this paper. The values of x/y and $\sqrt{x}$ will also be assumed to be integers. Algorithms and program examples will be described by a FORTRAN like notation, the meaning of which should be apparent.

2. Boolean Recurrence in Conditional Cyclic Loops

Suppose we have a conditional cyclic loop like the following:

```
L:  DO 1  $i = 2, n$
        IF $\mathbf{e}(x_{i-m}, \ldots, x_{i-1})$   THEN $x_i = \phi_i$
                                  ELSE $x_i = \pi_i$
 1  CONTINUE
```

where ϕ_i and π_i are arbitrary functions. Although the expression $\mathbf{e}$ may include variables other than x_i $(1 \le i \le n)$, we use the notation of $\mathbf{e}(x_{i-m}, \ldots, x_{i-1})$ to highlight dependence cycles between the predicate and the assignment statements. Then loop L can be represented by the following set of equations:

$$x_i = \begin{cases} \phi_i & \text{if } b_{i-1}=1 \\ \pi_i & \text{if } b_{i-1}=0 \end{cases} \quad (2 \le i \le n)$$

where $x_2, \ldots, x_n$ are variables, b_{i-1} is a Boolean variable defined by a Boolean expression $\mathbf{e}(x_{i-m}, \ldots, x_{i-1})$ $(1 \le m < n)$, and 0 and 1 are Boolean constants. If ϕ_i and π_i are constants (i.e., known values before executing the loop), then L is a postfix-IF loop of size n, and if ϕ_i and π_i are linear recurrences, then L is a linear mixed recurrence loop of size n. In the equation form of loop L, the value of the Boolean variable b_i depends on the values of some b_k's

($i-m \leq k \leq i-1$). So, every conditional cyclic loop has an imbedded Boolean recurrence.

Consider a set of Boolean variables $\{b_1, \ldots, b_i\}$ with an integer i. Let the 2^i minterms of these variables be numbered 1, 2, . . . , 2^i as they appear in a usual truth table, and $P_t(b_1, \ldots, b_i)$ be the t^{th} minterm. Then the Boolean variable b_i in loop L can be represented by the following nonlinear Boolean recurrence of order m:

$$b_i = \sum_{t=1}^{2^m} e_{i,t} P_t(b_{i-m}, \ldots, b_{i-1}), \quad (2 \leq i \leq n).$$

where $e_{i,t}$ is the value of the Boolean expression **e** based on $P_t(b_{i-m}, \ldots, b_{i-1})$, and $\sum$ denotes Boolean sum. So, one can parallelize the loop by solving the Boolean recurrence in parallel. In [1-3], postfix-IF loops are parallelized by solving the Boolean recurrence caused by the loop.

Since our discussion on the complexity of a conditional cyclic loop is based on the *fan-in lemma* in [8] (see also [6]), we borrow the lemma before proceeding. The assumption that a processor can consume at most two operands at a time is used in the lemma and will be used throughout the paper.

Lemma 1 (fan-in lemma). Suppose a processor can do at most a single binary operation at a time. Then an expression that depends on n variables or constants cannot be evaluated in less than log n time with an unlimited number of processors.

To solve the Boolean recurrence for b_i, the coefficients $e_{i,t}$'s need to be evaluated first. For a postfix-if loop, all the $e_{i,t}$'s can be evaluated in parallel without any difficulty, because all the possible values of x_i's are known. However, in a linear mixed recurrence loop of order m, the evaluation of $e_{i,t}$'s is considerably more complex than a postfix-IF loop. To evaluate the coefficients of a Boolean recurrence, we need to solve linear recurrences, and we need to solve the Boolean recurrence to determine the coefficients of linear recurrences. The straightforward way of breaking this circular nature, which we adopt, is to evaluate all the possible values of the recurrence variables. This forces us to solve a full-order Boolean recurrence:

$$b_i = \sum_{t=1}^{2^{(i-1)}} e_{i,t} P_t(b_1, \ldots, b_{i-1}), \quad (2 \leq i \leq n).$$

However, by the fan-in lemma, we cannot solve this full-order Boolean recurrence in o(n) time, and *a fortiori* O(log n) time, because there are $\Theta(n \cdot 2^n)$ variables in the expression to be evaluated. Notice that there are 2^{i-1} possible values of x_i for each i ($1 \leq i \leq n$).

3. Binary Trees and Conditional Cyclic Loops

We consider another approach of parallelizing a general conditional cyclic loop, which is based on a binary tree representation. Our concern is to parallelize conditional cyclic loops in general, which include both postfix-IF loops and linear mixed recurrence loops, on shared memory parallel computers allowing concurrent reads (the CREW PRAM [9]). For postfix-IF loops, our result is equivalent to the Boolean recurrence solver in [2]. However, our approach shows parallelism more clearly and is more general.

Consider loop L again. By making each node of a binary tree represent each possible value of the variables defining **e** and the two edges from each node represent the two branches of the IF statement, say the left edge for the false branch and the right edge for the true branch, loop L can be naturally represented by a binary tree of height $n-1$.

Consider the complete binary tree of height $n-1$. Let $e_{i,t}$ be the t^{th} node from the left on the i^{th} level of the tree (see Figure 2). Then $e_{i,t}$ ($1 \leq t \leq 2^{i-1}$) represents one of 2^{i-1} possible values of the variables defining **e** at the i^{th} iteration of loop L. So, the execution of a conditional cyclic loop L is equivalent to forming a certain path from the root by selecting a node at each level of the tree, provided that the tree is already formed.

We now consider solving the Boolean recurrence imbedded in a conditional cyclic loop by selecting a path on the tree, which is basically a parallel prefix problem. Let $Path_t$ ($1 \leq t \leq 2^{n-1}$) be the Boolean product of all the $e_{i,t}$'s on the path from the root to the t^{th} leaf node. Then, the problem is to find a $Path_t$ having Boolean value 1, which represents the path to be taken in executing the loop (there can be only one $Path_t$ having Boolean value 1).

Suppose a processor is assigned to each "mutually exclusive" complete subtree of height 2 of the binary tree from top to bottom, i.e. processors are assigned to the nodes on every other level of the tree starting from the root. By checking the value of the root of the subtree, each processor can determine which one of its two descendant nodes will be taken for the path we want to find. This produces $(2^n-1)/3$ edges for the tree of height n. Now, we want to form a reduced tree. Suppose we have two edges, say E_1 and E_2. If in the original tree, the parent node in E_2 is a left (right) son of the descendant node in E_1, then E_2 becomes a left (right) son of E_1. So, we have a tree whose height is half of the original tree's height (see Figure 3). Notice that this tree reduction is essentially the step of Boolean product in the parallel prefix problem. Doing this recursively until the tree is reduced to a single node, we can obtain the single path that a sequential execution of the loop follows in O(log n) time (see Algorithm 1). The correctness of Algorithm 1 can be easily checked by induction, and this leads us to the following lemma.

Lemma 2. In a conditional cyclic loop of size n, the imbedded Boolean recurrence can be solved in O(log n) time with an unlimited number of processors.

To select a certain path in the tree representation of the loop (*path selection stage*) by using Algorithm 1, we need to set up the tree first by precomputing all the possible values of the variables defining the predicate in the loop (*precomputation stage*). For a linear mixed recurrence loop of order m and of size n, the precomputation stage is equivalent to solving 2^{n-1} linear recurrences of order m. Suppose we solve all the recurrences at a time by using a fast parallel recurrence solving algorithm like the one in [4] or [9] because we expect "small" m in practice. Then, assuming that the Boolean expression **e** can be evaluated in constant time if the values of the variables defining it are immediately available, the recurrences can be solved in approximately $(2 + \log m) \log n$ time with an unlimited number of processors. So, we have the following theorem.

Theorem 3. Any arbitrary linear mixed recurrence loop of order m and of size n can be executed in O(log m log n) time with an unlimited number of processors.

In a postfix-IF loop the number of possible values of the predicate is determined by the order of a postfix-IF loop. There are at most 2^m nodes at each level of the tree for a postfix-IF loop of order m. Since all the possible values are immediately available, we have the following corollary directly from Lemma 2, by taking the first 2^m nodes at each level from the complete binary tree in Figure 2. Notice that all the $e_{i,u}$ result the same value of b_i as $e_{i,j}$ where $j = (u-1) \bmod 2^{m-1} + 1$, because the value of each $e_{i,u}$ depends only on the values of $e_{l,t}$'s $(i-m \leq l \leq i-1)$.

Corollary 4. A postfix-IF loop of order m and of size n can be executed in $O(\log n)$ time with an unlimited number of processors.

4. With A Limited Number of Processors

Since p should be a relatively small, limited number in practice, we consider a way of exploiting the parallelism in a conditional cyclic loop with a limited number of processors. Consider the binary tree in Figure 2 again. To use Algorithm 1, the tree is partitioned by level so that each partition covers $\log(3p+1)$ levels. By assigning processors to the first $\log(3p+1)$ levels of the tree, we can find the first $\log(3p+1)$ nodes for the path we want to have. The next $\log(3p+1)$ nodes for the path can be found in the same way by considering a subtree with its root as the last one of the first $\log(3p+1)$ nodes found for the path. By applying Algorithm 1 to each partition iteratively in this way, the path selection stage can be done in approximately $\frac{(n-1)\log\log(3p+1)}{\log(3p+1)-1}$ time. Notice that Algorithm 1 is equivalent to the sequential execution of a conditional cyclic loop when $p = 1$. This gives us the following lemma and corollary.

Lemma 5. The path selection stage for a conditional cyclic loop of size n can be solved in $O(\frac{n \log\log p}{\log p})$ time with p processors.

Algorithm 1
(Path selection for a conditional cyclic loop)

```
  /* the value of every $e_{i,t}$ is known */
  /* $P_{i,t}$ is an ordered set of nodes */
  /* $P_{1,1}$ is the output */

L1: DO 1 $k = 1, \log(n-1)$
L2:   DOALL 2 $j = 1, (n-1)/2^k$
          $i = (j-1)2^k + 1$
L3:     DOALL 3 $t = 1, 2^{i-1}$
          IF $(k = 1)$
            THEN IF $e_{i,t}$ THEN $P_{i,t} = \{e_{i,t}\} \bigcup \{e_{i+1,2t}\}$
                              ELSE $P_{i,t} = \{e_{i,t}\} \bigcup \{e_{i+1,2t-1}\}$
            ELSE BEGIN
              $e_{i+2^{k-1}-1,x}$ = the last entered element of $P_{i,t}$
              IF $e_{i+2^{k-i}-1,x}$ THEN $P_{i,t} = P_{i,t} \bigcup P_{i+2^{k-1},2x}$
                                ELSE $P_{i,t} = P_{i,t} \bigcup P_{i+2^{k-1},2x-1}$
                END
 3          CONTINUE
 2        CONTINUE
 1      CONTINUE
```

Corollary 6. A postfix-IF loop of order m and of size n can be executed in $O(\frac{2^m n}{p} \log \frac{p}{2^m})$ time with p $(> 2^m)$ processors.

Since in practice it is rare for m to be greater than three, we may consider 2^m to be a constant. So, a postfix-IF loop can be executed in $O(n/p \log p)$ time.

As noted earlier, the precomputation stage adds considerable complexity in parallelizing a linear mixed recurrence loop. To get a path of length $\log(3p+1)$ we need to form a tree of height $\log(3p+1)$ for a linear mixed recurrence loop. This requires us to solve $(3p+1)/2$ linear recurrences of order m and of size $\log(3p+1)$. Since there are at most $O(p)$ nodes in a tree of height $\log(3p+1)$, the precomputation stage for a partition can be done in $O(m^2 \log \frac{\log p}{m})$ time by using the algorithm in [5].

Theorem 7. A linear mixed recurrence loop of order m and of size n can be solved in $O(\frac{m^2 n}{\log p} \log \frac{\log p}{m})$ time with p processors $(p \geq 2m)$.

Unfortunately, the benefit of the parallelization is little, which makes it impractical parallelizing linear mixed recurrence loops in general.

5. Special Cases of Linear Mixed Recurrence Loops

Although we are pessimistic about the existence of an 'efficient' parallel solution for a general linear mixed-recurrence loop, we have three interesting special cases of order 1. Consider a linear mixed recurrence loop of order 1, which can be represented by the following equation:

$$x_i = \begin{cases} a_i \cdot x_{i-1} + c_i & \text{if } b_{i-1} = 1 \\ \overline{a}_i \cdot x_{i-1} + \overline{c}_i & \text{if } b_{i-1} = 0 \end{cases}$$
$$(2 \leq i \leq n)$$

where a_i, c_i, $\overline{a}_i$, and $\overline{c}_i$ are coefficients. Then we have the following three special cases:

Case I:	$c_i = 0$, $\overline{c}_i = 0$, a_i are fixed for all i, and $\overline{a}_i$ are fixed for all i
Case II:	$a_i = 1$, $\overline{a}_i = 1$, c_i are fixed for all i, and $\overline{c}_i$ are fixed for all i
Case III:	Either $a_i = 0$ for all i or $\overline{a}_i = 0$ for all i (0 and 1 are integer)

Although our special cases seem to be quite restricted, it is interesting to observe that the special cases are not rare in practice (in our experiments of automatic program restructuring [7], most linear mixed recurrence loops are of special cases). Notice that all the special cases are recurrences with constant coefficients. Furthermore, the number of possible values of x_i is reduced. By the commutativity of multiplication (for Case I), by the commutativity of addition (for Case II), and by induction (for Case III) there are i possible values of x_i for each i $(1 \leq i \leq n)$. This reduced number of possible values and the constant coefficients naturally simplify the precomputation stage and the path selection stage.

By partitioning the binary tree by level so that each partition has $\sqrt{p}$ levels, and by finding a subpath for each partition iteratively starting from the first partition, we have the following corollary.

Corollary 8. The three special cases of linear mixed recurrence loops can be executed in $O(\frac{n \log p}{\sqrt{p}})$ time with p processors.

6. Conclusion

We have considered parallelizing conditional cyclic loops on a shared memory multiprocessor allowing concurrent reads. Based on a binary tree representation of a conditional cyclic loop, executing the loop turns out to be equivalent to precomputing all the possible values of variables involved in defining predicates of the loop (precomputation stage) and selecting a single path from the root of the tree (path selection stage).

Although little benefit of the parallelization with a finite number of processors makes it impractical to parallelize linear mixed recurrence loops in general, postfix-IF loops do not require the precomputation, and the special cases of linear mixed recurrences loop reported require the time for precomputation increasing quadratically with respect to the loop bound n. Furthermore, the path selection stage for these loops can be done easily because of reduced number of nodes in the tree. Postfix-IF loops and the special, but not rare, cases of linear mixed recurrence loops can be executed in $O(\log n)$ time with a polynomial number of processors with respect to the loop bound n.

The difficulty of parallelizing conditional cyclic loops emphasizes the importance of developing entirely new parallel algorithms. However, developing an efficient new parallel algorithm often requires long hard work and some ingenuity. In the mean time, the way of parallelizing conditional cyclic loops presented in this paper can be used, by identifying a conditional cyclic loop automatically, for a "modest" gain of speedups from some automatically restructured sequential programs.

```
      DO 1 i = 1, n
        IF (y(i) .GT. 0)
        THEN x(i) = a(i) + b(i)
        ELSE x(i) = a(i) * b(i)
1       CONTINUE
```

a). conditional acyclic loop

```
      DO 1 i = 2, n
        IF (c(i) .GT. c(i-1))
        THEN c(i+1) = w(i+1)
        ELSE c(i+1) = v(i+1)
1       CONTINUE
```

b). postfix-IF loop of order 2

```
      DO 1 i = 2, n
        IF (x(i-1) .GT. 0)
        THEN x(i) = x(i-1)*2 - t
        ELSE x(i) = h(i)
1       CONTINUE
```

c). linear mixed-recurrence loop of order 1

Figure 1. Loops with Conditional Branching

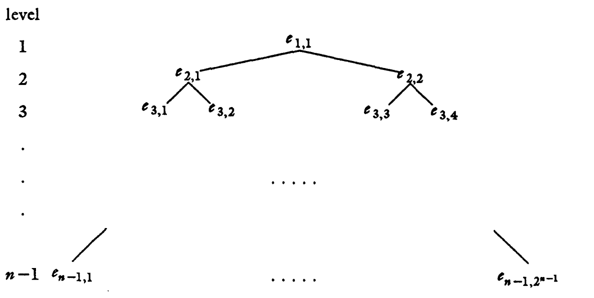


Figure 2. A Tree Representation of a Conditional Cyclic Loop

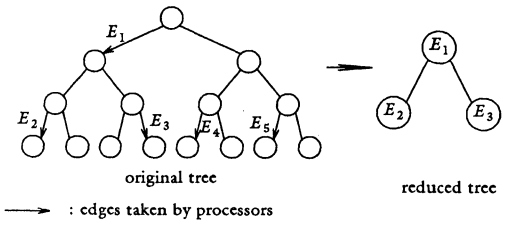


Figure 3. Tree Reduction

References

[1] U. Banerjee, "Speedup of Ordinary Programs", Ph.D Thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1979

[2] U. Banerjee and D. Gajski, "Fast Evaluation of Loops with IF statement", *IEEE Trans. on Computers,* Vol. C-33, No. 11, pp. 1030–1033, 1984

[3] U. Banerjee, D. Gajski, and D. Kuck, "Array Machine Control Units for Loops Containing IFs", *Proc. of the 1980 International Conf. on Parallel processing,* Aug. 1980

[4] S. Chen, "Speedup of Iterative Programs in Multiprocessor Systems", Ph.D Thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1975

[5] S. Chen, D. Kuck, and A. Sameh, "Practical Band Triangular System Solvers", *ACM Trans. on Mathematical Software,* Vol. 4, No. 3, pp. 270–277, Sept. 1978

[6] D. J. Kuck, *The Structure of Computers and Computations,* Vol. I, John Wiley & Sons, Inc., NY, 1978

[7] G. Lee, C. Kruskal, and D. Kuck, "An Empirical Study of Automatic Restructuring of Nonnumerical Programs for Parallel Processors", *IEEE Trans. on Computers,* Vol. c-34, No. 10, 1985, pp.927–933

[8] I. Munro and M. Patterson, "Optimal algorithms for parallel polynomial evaluation", *J. Comput. Syst. Sci.,* 7(1973)

[9] A. Sameh and R. Brent, "Solving Triangular Systems on a Parallel Computer", *SIAM J. Num. Analysis,* Vol. 14, No. 6, pp. 1101–1113, Dec. 1977

[10] M. Snir, "On Parallel Search", presented at the Ottawa Conf. Distributed Computing, Aug. 1982

[11] M. J. Wolfe, "Optimizing Super Compilers for Supercomputers", Ph.D Thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1982

Debugging Parallel Programs using Graphical Views*

Mary L. Bailey, David Socha, and David Notkin

Department of Computer Science, FR-35
University of Washington, Seattle, WA 98195

Abstract

Voyeur is a prototype system for creating application-specific, graphical views of parallel programs. We describe the system and three views created using the system, two for MIMD non-shared memory parallel programs and one for a shared memory, multi-threaded program.

1 Introduction

Historically, computers have supported debugging by providing access to the program's state. The programmer assimilates this information, comparing the expected and actual states of the computation to validate the program's execution or to detect errors. This approach often is overwhelming for parallel programs, which have orders of magnitude more state information than sequential programs. Graphical views help to manage this state information by synthesizing images of the program's state and thus focusing on the problem structure, algorithmic structure, or architectural structure of the target computer. These images present a great deal of information in a readily assimilated manner.

Voyeur is a prototype system for constructing application-specific graphical views of parallel programs. Voyeur's goal is to make practical the creation of new views for specific algorithms. To date, we have constructed and used Voyeur views of (1) MIMD, non-shared memory programs written in Poker [12], (2) a shared memory parallel simulation program executed on a Sequent multiprocessor, and (3) a sequential Fortran program executed on a MicroVAX-2. This paper describes three of these Voyeur views and discusses their use. We also describe how Voyeur provides a structure for easily creating views.

Related Work. Techniques to visualize the data coming from parallel debugging range from using a sequential debugger on each of the processes of a parallel program [11] to providing a textual trace of program execution [8] to integrating trace information and a visual view of the program. The last category includes Belvedere [7], which displays the communication graph of a message passing program and shows the message activity on the edges of this graph, and also can find and display logical patterns of message activity in an asynchronous message-passing system. Similarly SDEF [5] and Poker's Trace View [12] display variable values in the nodes of a communication graph.

2 Voyeur Views

This section describes three Voyeur views and how they helped verify and debug parallel programs. The three views were developed for different applications: two Poker programs and a shared memory, multi-threaded program. Several additional views have been developed [13] but are not discussed here due to space limitations.

Icon View. The icon view (see Figure 1) was developed to debug a sharks and fishes algorithm [4]. The algorithm simulates sharks and fishes moving in a two-dimensional grid of points. Each point may be occupied by one animal. Fish move into vacant, randomly chosen, adjacent points. Sharks move similarly, with the exception that sharks eat an adjacent fish if possible. Both species occasionally give birth, with the baby staying in the place the parent vacates. Fish never starve, but sharks do if

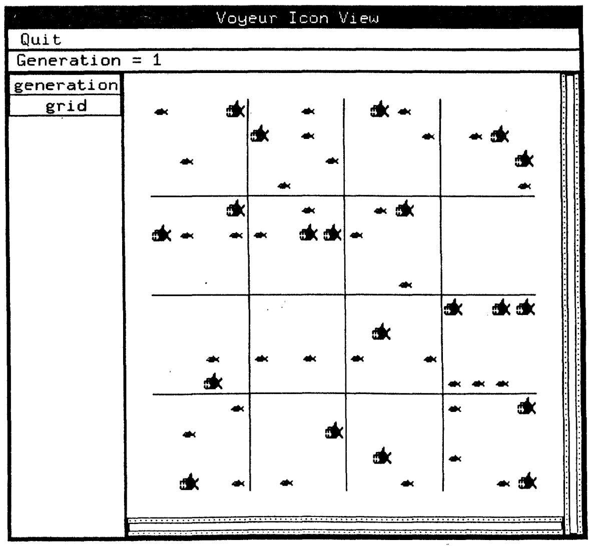


Figure 1: Voyeur's icon view.

*This research funded in part by Office of Naval Research Contract N00014-86-K-0264, National Science Foundation Grant CCR-8416878, and Air Force Office of Scientific Research Contract 88-0023.

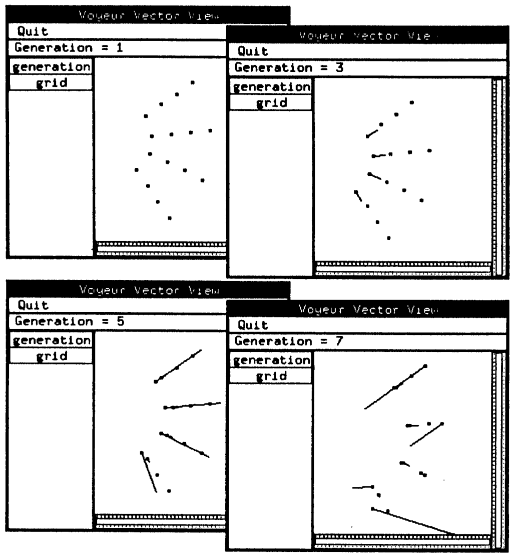


Figure 2: Voyeur's vector view.

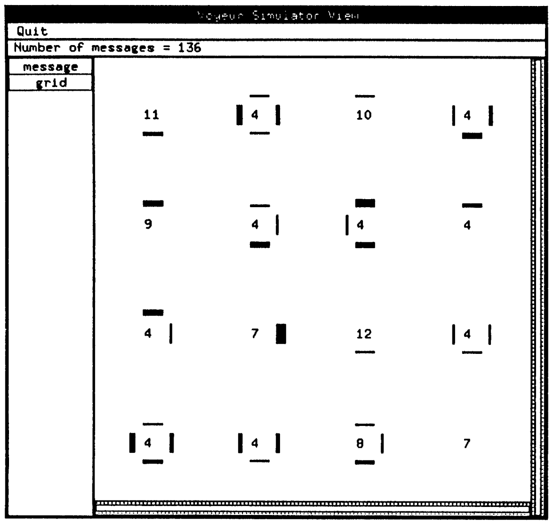


Figure 3: Voyeur's simulator view.

left unfed long enough. For simplicity, all the fish and then all the sharks are moved, in an alternating pattern.

Debugging this algorithm proved difficult because of the large amount of information in each processor, the synchronization between processors, and the random nature of the algorithm. For instance, when testing the movement of the fishes, having the two fish in a single processor randomly choose to move west seemed fine until the icon view showed all of the fish "randomly" moving west – clearly a bug.

Another bug manifested itself in the communication between processors. The two-dimensional problem space was divided among the processors which were connected in a mesh (the grid lines are shown in Figure 1). At one point in the program, *some* of the fish on the east side of a processor jumped to the west side of the same processor, and vice versa. This was due to a program error: the constants `east` and `west` had been reversed. While this bug could have been detected using only local information, the global view made it obvious.

Vector View. The vector view plots vectors in a two-dimensional space. This view was created to assist a colleague in the Applied Math department debug a Poker program of the SIMPLE algorithm [2, 6]. The calculation was going wild and two causes were possible: either there was a bug in the Poker program, or there was a numerical instability due to the sparseness of the points in the 3-dimensional space. Viewing the vectors during the program's execution, the programmer saw a vector move non-radially before the errant behavior, indicating numerical instability, rather than a program bug (see Figure 2).

We developed this view by first generalizing and then specializing the icon view. Each line of a setup file specifies the type of an object to view (icon or vector), the simulator's unique identifier for the species of object (such as a shark icon or a hydro vector), and, for icons, a file containing the bitmap for that object species. Adding or removing species of objects and changing their appearance is as easy as modifying this file.

Simulator View. The simulator view, developed for a colleague in Computer Science, monitors a parallel simulation of message flow among 16 processes connected in a 4×4 torus. Each process is connected to its four neighbors. When a process receives a message, it updates its local clock and forwards the message, with the new clock time, to a randomly chosen neighbor. The run-time system guarantees that messages are delivered in timestamp order, so that processes can never receive old messages (messages with timestamps less than the current process's clock). The simulation is written in SYNAPSE [14] and runs on a Sequent [10] multiprocessor.

Figure 3 shows the message traffic of the system. Each process shows its local time and the length of the queues of incoming messages as bars. For instance, the process in the upper right corner is at time 4 and has one message in the west queue, two in the east queue, and three in the south queue.

At the beginning of the simulation, each process generated four messages and sent them out in random directions. No other messages were generated during the simulation. The uniformity of the queue length and clock times reaffirmed our intuition about the effects of randomly selecting message ports in this regular interconnection structure. Modifying the vector view to create the simulator view took five hours.

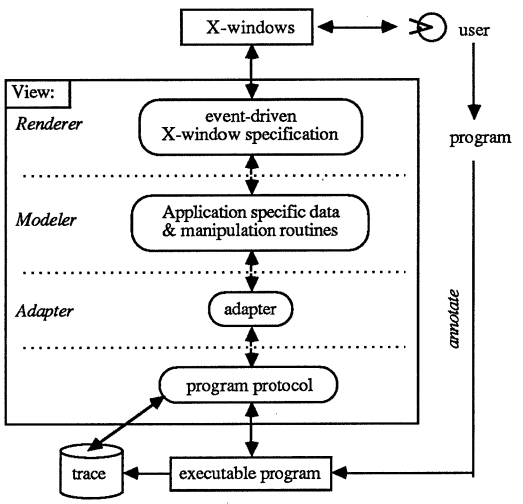


Figure 4: Voyeur system structure

3 Integrating Voyeur and Programs

The various views has been invaluable in debugging Poker programs; however, the key to Voyeur's usefulness is the ease of creating new views. This is facilitated by Voyeur's structure (shown in Figure 4) and a hierarchy of views [13]. Boxes with square corners are heavy-weight processes. Boxes with round corners are modules. Messages from the user filter down to change the form of the view or to request more simulation data. Messages from the simulator filter up to change the state shown by the view.

A Voyeur view consists of the simulator interface, the adapter, the modeler, and the renderer [1]. The adapter translates between the string-based simulator messages and the procedural interface of the modeler and renderer. These messages may come directly from the simulator, or may come from a trace file produced by the simulator. Based on the type of each simulator message, the corresponding modeler procedure for that message is called. The modeler manages data specific to the application. The renderer defines the user interface (based on X Windows [9]), which is responsible for drawing the view of the information in the modeler, for manipulating the form of the view, and for letting the user control the program's execution. Just as control events from the X Window interface drive the execution of the renderer and modeler, state messages from the simulator drive the adapter, the modeler, and the renderer.

The user interfaces of the views share a basic structure. The view's title is contained in a title bar at the top of the view. Underneath the title bar is a set of pull-down menus. Below the menu bar is a status area containing data appropriate for the view. Below this and to the left is a set of control buttons for controlling the execution of the simulation. The data area, in the lower right-hand corner, contains scroll bars for movement within this area.

To create a new view, the user first annotates the Poker program to send appropriate messages to the view. The adapter is compiled from a description of these messages. The user then writes the modeler and the renderer. Because the view share much of their functionality and appearance, creating a new view consists of modifying an existing one. This typically takes anywhere from a few hours to a few days. We hope to shorten this time by converting Voyeur from C to C++ and explicitly utilizing the class hierarchy.

4 Conclusions

The Voyeur prototype provides easy construction of new and flexible views for parallel debugging. These views have greatly eased the laborious task of finding obscure bugs in Poker programs. The current structure has a fairly high degree of flexibility both in the power of the views and in the creation of new views.

Still, there are many areas for improvement; we are now pursuing these as part of the *Orca* project. We need to simplify the task of creating new views by techniques such as an explicit hierarchy of views. We need to explore new views. We need to increase the flexibility of the existing views. For instance, in the icon view, it would be nice to allow the user to dynamically select which icons to see. Similarly, logical zooming is a powerful tool. For example, zooming out using the icon view could replace the icons with smaller icons and eventually just a dot for each icon, progressively giving a larger global picture at the expense of local information. Finally, we need to explore using Voyeur with more programming systems.

References

[1] Marc H. Brown. *Algorithm Animation.* PhD thesis, Department of Computer Science, Brown University, April 1987.

[2] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE Code. Technical Report UCID-17715, Lawrence Livermore Laboratory, February 1978.

[3] dbx. In *UNIX User's Manual Reference Guide.* 4.2 Berkeley Software Distribution, USENIX Association, March 1984.

[4] A. K. Dewdney. Computer Recreations. *Scientific American*, pages 18–24, July 1984.

[5] B. R. Engstrom and P. R. Capello. The SDEF Systolic Programming System. In Sartaj K. Sahni, editor, *Proceedings of the International Conference on Parallel Processing*, pages 645–652, St. Charles, IL, August 1987.

[6] Kevin Gates. SIMPLE, an Exercise in Programming in Poker. Technical Report 88-2, Department of Applied Math, University of Washington, March 1988.

[7] Alfred A. Hough and Janice E. Cuny. Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation. In Sartaj K. Sahni, editor, *Proceedings of the International Conference on Parallel Processing*, pages 735–738, St. Charles, IL, August 1987.

[8] Terrence W. Pratt. The PISCES 2 Parallel Programming Environment. In Sartaj K. Sahni, editor, *Proceedings of the International Conference on Parallel Processing*, pages 439–445, St. Charles, IL, August 1987. IEEE Computer Society Press.

[9] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[10] S. S. Thakker, P. R. Gifford, and G. F. Fieland. Balance: A Shared Memory Multiprocessor System. In *Proceedings of the 2nd International Conference on Supercomputing*, May 1987.

[11] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–23, January 1985.

[12] Lawrence Snyder. Parallel Programming and the Poker Programming Environment. *IEEE Computer*, 17(7):27–36, July 1984.

[13] David Socha, Mary Bailey, and David Notkin. Voyeur: Graphical Views of Parallel Programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WI, May 1988.

[14] David D. Wagner, Edward D. Lazowska, and Brian N. Bershad. Techniques for Efficient Shared-Memory Parallel Simulation. Technical Report 88-04-05, Department of Computer Science, University of Washington, April 1988.

An Integrated and Portable Tool Environment for Parallel Computers

Thomas Bemmerl

Technical University Munich, Department of Computer Science
Laboratory for Parallel Computing
Arcisstr. 21, D–8000 München 2, Federal Republic of Germany

Abstract

The paper is concerned with the problem of designing tools for multiprocessors. The features, the design concepts and some implementation details of an integrated and portable tool environment for multiprocessors are presented. The tool environment MMS (Multiprocessor Monitoring System) contains tools for debugging, performance analysis and visualization of multiprocessors and their program execution. Apart from the functionality of the tools, MMS offers the following features; portability to various parallel architectures, expandability and adaptability with new tools and languages, and support of several abstraction levels. In addition, the tool environment is based on different instrumentation and monitoring techniques. The main design concept of MMS is a new hierarchical layered model for tool environments, which will be presented in the paper.

1. Introduction and State of the Art

Today, programs for many parallel machines are written in a high level programming language and are based on the objects and system calls of the concurrent operating system of each processor node. In addition, software for many parallel computers is developed in a so called host/target environment. This means, that the programs are developed with cross–compilers on a host computer and are downloaded for execution into the target system. The host is normally a conventional workstation connected with the target multiprocessor via parallel busses or LAN's. Many supercomputers from industry and university are programmed this way, for example; the iPSC of Intel, the Cosmic Cube, the NCUBE, the Pringle Parallel Computer, the Mark II/III etc. In addition, even multiprocessors with native–compilers include central development nodes, from which programs are downloaded into the processor elements [4]. Therefore a "logical" host/target environment is at least existing within this class of multiprocessors.

Tools for generating parallel programs, i.e. compilers and linkers, are already available for multiprocessors. In contrast, very limited possibilities are at hand for looking at the dynamic behavior of parallel computers and their software. No adequate tools are available for debugging, performance measurement and visualization of program execution. Parallel programming environments available from university and industry have several disadvantages with respect to debugging, monitoring, instrumentation, performance measurement and visualization:

* Some tools are not integrated into the programming environment. Therefore monitoring is done at a low abstraction level without any relation to the programming concepts [4]. These tools mainly focus on instrumentation techniques and synchronization concepts for multiple monitors.
* Other projects are mainly interested in programming concepts. In these projects monitoring is done using source code instrumentation, operating system instrumentation and runtime instrumentation [9], [6], [7]. These concepts lead to more batch oriented tools because the insertions into the source code have to be recompiled each time the programmer needs new information about program execution. In addition, the inserted monitoring instructions modify the runtime behavior of the multiprocessor. This has awkward influences on the synchronization behavior of application programs.
* The third class of tools focuses mostly on specific architectures of parallel computers. There are, for example, tools available for bus–oriented multiprocessors. The monitoring techniques of these tools depend on the availability of bus–oriented connections between the processor elements [3].

Motivated by these shortcommings of the available tools, two years ago we started the design and development of an expandable and portable tool environment for concurrent computers. The environment contains integrated tools for debugging, performance analysis and visualization of the dynamic behavior of concurrent multiprocessors and their program execution. MMS is usable for program development, program optimization, for studying existing programs and for teaching the dynamic behavior of parallel computers and concurrent programs. Before I describe the design concepts and the state of the project, I will define in the next paragraph the requirements of tool environments for parallel computers.

2. Features and Requirements

The following described requirements are met by our tool environment MMS. Therefore the explanation of the requirements can also be considered as presentation of MMS features. I first explain first in a coarse–grained manner the functionality of the available frontends of the environment, the single tools.

2.1. Functionality of the Tools

The first tool, mainly used for program development, is a window based concurrent debugger. The debugger offers features for displaying and modifying the states of programs running on the multiprocessor [8]. A powerful debugging language is available, which allows the specification of complex predicates about the dynamic execution of programs [8], [1]. Based on these conditions, several actions are initiated, e.g. stopping processes (breakpoints) or recording state changes of the program (tracing).

The second tool is a performance analyzer for optimizing concurrent programs [5]. Using the performance analyzer, the programmer gets information about the efficiency of the communication between processes or processor elements, about the activation of procedures, the access to variables and operating system objects. In general, the performance analyzer records the access to various objects and displays these activities using several kinds of charts. The performance analyzer gives the user the ability to localize the bottle–necks of his implementation. Based on this information, the programmer can develop a optimized mapping of processes onto processors.

Another possibility for displaying the dynamic behavior of multiprocessors is to use the visualization tool. This part of the tool environment gives users the ability to look in a graphical manner at the dynamic execution of programs. This tool shows on a graphic display the flow of communication between processes or processors. Additionally, the visualization tool displays complex data types, the control and the data flow of programs or processes in a graphical way. In general, this tool gives a graphical representation of the multiprocessors state space.

The tool environment is not limited to interactive tools. MMS is also expandable with more batch oriented tools, like systems for automatic testing. These frontends of the tool environment offer possibilities for regression tests and C1–tracing which are necessary for program testing and maintenance. However, this is not a main focus of our project.

2.2. General Requirements and Features

An important disadvantage of existing tools, mentioned during the introduction, is their isolation, lack of flexibility, portability and expandability. The following discussions concern these more general requirements of tool environments for multiprocessors. An important requirement of tool environments is their frequent interactive usage. Tools like debuggers are used very interactively during program development. Therefore a unique and user friendly human interface with graphic support, multiple windows, menus and mouse interaction is offered by MMS.

All described tools need information about the dynamic execution of programs on the multiprocessor. This information can be collected using different instrumentation and monitoring techniques. Monitoring techniques considered in our project are hardware instrumentation, object code instrumentation (software instrumentation) and hybrid instrumentation. All these instrumentation techniques support interactive usability of the tool environment. No extra recompilation for monitoring is necessary. An important feature is that the single tools need not know the instrumentation technique they are actually based on. Therefore different monitoring techniques can be used across different processor elements.

A very important requirement is the abstraction level of the tools [1], [5]. The programmer must be able to use the tools at different abstraction levels, from a very low abstraction level (e.g. machine level or assembler) up to a very high abstraction level (e.g. process level). At the low level, the tool environment handles addresses and data types of the processor. At the high level, MMS knows all objects of the programming language (e.g. procedures and variables) and all objects of the concurrent operating system (e.g. tasks, semaphores, mailboxes) by their names. All objects are specified and displayed with the syntax and semantic of the selected abstraction level. The user has the ability to use MMS at the described abstraction levels (machine level, C language level and process level). This means that one programming construct can be inspected at different abstraction levels.

Another requirement focuses on portability. A tool environment for host/target environments has to support various target architectures. Therefore the target parts of MMS are easily portable to various different multiprocessor architectures. The tool environment is not only based on "real" parallel computers. An instruction level simulation of the target system on the host computer may also be sufficient for locating most logical errors. Once more, the single tools need not know whether they are based on real hardware or only on the simulation of the real hardware.

A further disadvantage of state of the art tools is their weak integration. All tools of our environment are based on the same instrumentation techniques and they offer a unique human interface. In addition, for an integration it is necessary for all tools to use the same symbol data base and the same symbol translation mechanism.

The last general requirement concerns the expandability and the adaptability of the tool environment. This requires that the environment is expandable with new frontends (tools) without a modification of the instrumentation techniques (monitors). For example, the monitors have to be powerful enough to add an automatic test system as new frontend without changing the monitors. Additionally, MMS is easily adaptable to new programming environments with new programming concepts and compilers. This requires an easy adaptation of the symbol data base and the monitoring of the stack structures to new requirements.

3. Design Concepts of MMS

In order to satisfy the presented requirements, an ingenious system is necessary which consists of hardware and software and is divided between host and target. The various tasks, requested by the requirements, are only performable by a very flexible design concept. The base of our design concept is a new hierarchical layered model for tool environments. The model is illustrated in figure 1. In analogy to the idea of the ISO/OSI model for network architectures, a layer of level i needs no knowledge of modules installed at levels lower than i. Therefore several implementations of one layer are replaceable without complications. This characteristic makes the tool environment adaptable to the various described requirements. For example, the hardware monitor is easily replaceable by a software monitor without changing the upper layers of the model. Additionally, MMS is expandable with new tools without changing the layers below S4. In general, a tool environment based on the hierarchical layered model meets the requirements of portability, expandability and adaptability.

For understanding the layered model, I explain in the following paragraphs the functionality of the several layers in a bottom–up manner. Three different monitors are available for monitoring the dynamic behavior of each processor element. The monitors are based either on hardware instrumentation, on software instrumentation (object code instrumentation) or on hybrid instrumentation. The monitors evaluate the predicates specified about the dynamic program behavior. There are four classes of predicates to evaluate; predicates about the control flow, predicates about the data flow, predicates about concurrency objects (e.g. tasks, mailboxes, semaphores) and predicates about combinations of the previous ones. It should be noted that all three monitors offer the same interface to the upper layers. They only differ in the retardation of program execution, but not in their functionality.

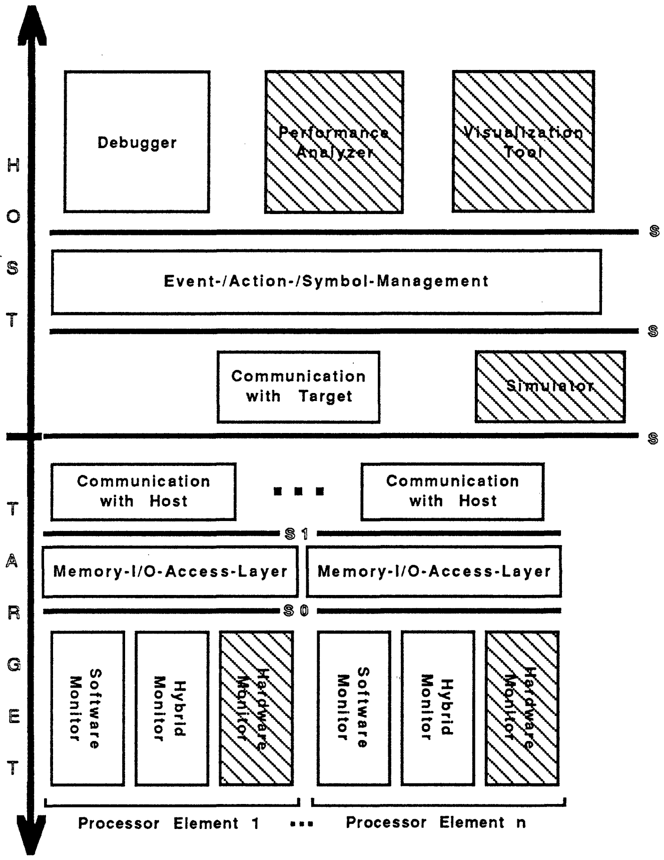


Fig. 1: Hierarchical Layered Model for Multiprocessor Tool Environments

The memory–I/O access layer implements the access to memory cells and I/O ports. Therefore this layer is responsible for displaying and modifying the contents of memory cells and I/O ports. In addition, this layer displays and modifies objects of the concurrent operating system. The target layers of each processor element are in general a concurrent assembler oriented monitoring tool based on physical or virtual addresses and processor data types. As already mentioned, even this low abstraction level is accessible by the user of MMS.

The simulator offers the same functionality as the target layers. Therefore the tool environment is usable without the availability of the target system. This means, that the simulator implements commands for accessing memory and I/O of the simulated multiprocessor. Additionally, the simulator allows the specification of low level predicates about the simulated program execution.

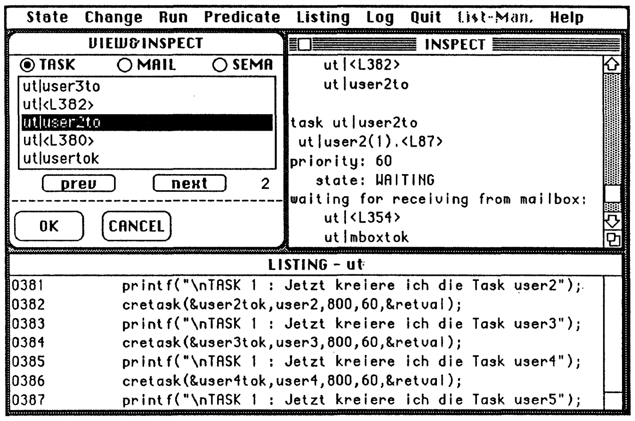
The central layer at the host workstation is the layer for event, action and symbol management. This layer offers primitive events, actions and a mechanism for combining these primitives. Based on this event and action mechanism the layers on top of the model (the tools) are able to specify complex predicates about the dynamic behavior of the target system [1]. The specification of events and actions is based on symbols used in the program to be monitored. This layer is also responsible for translation and retranslation of symbols into addresses and vice versa. Because this task requests several tedious algorithms for sorting and searching the symbol management is based on an adequate data base.

For the explanation of the top layer's functionality (debugger, performance analyzer, visualization tool) I refer to chapter 2.1. These tools are based only on the event and action concept of the layer below them (S4). Therefore these tools do not know the instrumentation and monitoring technique they are actually based on.

4. State of the Project

In chapter 1 I mentioned that the tool environment has been in development for about two years. In the following I describe the state of the project and future work.

Since summer 1987 the first tool of the environment, the concurrent debugger, has been finished. For an illustration of the debugger functionality, figures 2 and 3 represent two typical printouts of the debugger user interface. Figure 2 gives an example of the view/inspect command. Using this feature, the programmer can inspect the objects of the concurrent operating system, e.g. the states of existing tasks. In figure 3 the specification of a predicate, in this case a breakpoint about the data flow, is illustrated. The availability of the debugger implies of course the disposability of the lower layers of the model, because the debugger is a layer at the top.



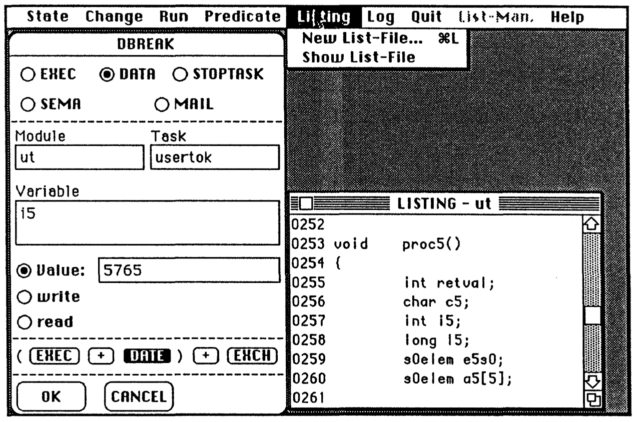
```
ut|user3to
ut|<L382>
ut|user2to
ut|<L380>
ut|usertok
2
```

```
    ut|<L382>
    ut|user2to

task ut|user2to
 ut|user2(1).<L87>
priority: 60
   state: WAITING
waiting for receiving from mailbox:
    ut|<L354>
    ut|mboxtok
```

```
printf("\nTASK 1 : Jetzt kreiere ich die Task user2");
cretask(&user2tok,user2,800,60,&retval);
printf("\nTASK 1 : Jetzt kreiere ich die Task user3");
cretask(&user3tok,user3,800,60,&retval);
printf("\nTASK 1 : Jetzt kreiere ich die Task user4");
cretask(&user4tok,user4,800,60,&retval);
printf("\nTASK 1 : Jetzt kreiere ich die Task user5");
```

Fig. 2: View/Inspect



Module: ut
Task: usertok
Variable: i5
[✓ Value:] 5765
write
read

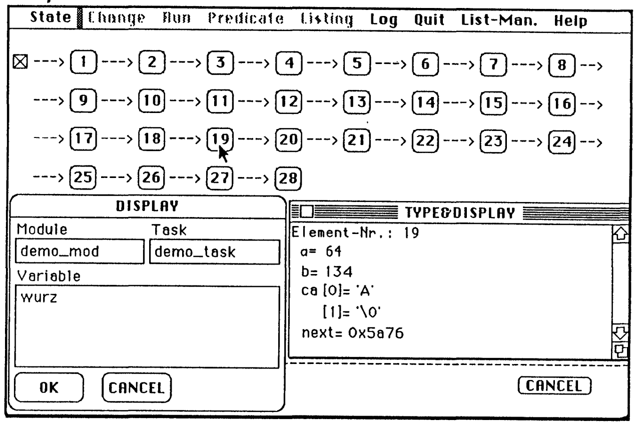
```
void    proc5()
{
        int retval;
        char c5;
        int i5;
        long l5;
        s0elem e5s0;
        s0elem a5[5];
```

Fig. 3: Breakpoint Specification

In addition to the debugger, the event, action and symbol management layer on the host workstation has been already finished. The host computer for this implementation was a VAXstation II with Ultrix32m, Xwindows and a cross-software development system. A C-cross-compiler is used for generating code for the 80186 microprocessor. The concurrent operating system for the target has been implemented as a library on the host which is linked to the application program. The operating system is object oriented and offers dynamically created tasks, mailboxes, semaphores and the usual operations (system calls) on these objects [2]. The concurrent operating system can be considered as a concurrent language extension of C.

The host layers are based on the existing target parts of MMS. The memory-I/O access layer as well as two different monitors are available. Therefore the host layers can be based on two monitoring techniques; on a hybrid monitor adapted to the processor bus of each processor element and on a software monitor based on object code instrumentation. The target layers have been implemented on a target system based on only one processor element. This processor element is an 80186 single board computer. The target system has been used as multiprocess (multitasking) singleprocessor system within the first prototype implementation of MMS. The already finished subsystems of MMS are illustrated in figure 1 with unhatched boxes.

Currently, the tool environment is adapted to target systems consisting of more than one processor element. An 32-node iPSC Personal Supercomputer from Intel is the first target system for the multiprocess multiprocessor implementation of MMS. The host layers of this implementation remain on the VAXstation II. We connected the host layers on the VAX via the TCP/IP network of the iPSC cube manager with the target layers in several nodes. Porting the target layers of MMS to the message passing hypercube machine is not very difficult because the nodes of the iPSC are based on an 80286/386 microprocessor which is a superset of the 80186. This retargetting of MMS on the iPSC will be finished in autumn 1988. For demonstration of the suitability of the tool environment for different multiprocessor architectures we plan in the future the retargetting of MMS on a memory coupled multiprocessor based on Motorola 68020/68881. In addition, the performance analyzer and the visualization tool are in the specification phase. A first approach for visualizing complex data types in a graphical manner has been already implemented within the concurrent debugger (see figure 4).



```
⊠ ---> 1 ---> 2 ---> 3 ---> 4 ---> 5 ---> 6 ---> 7 ---> 8 -->
  ---> 9 ---> 10 ---> 11 ---> 12 ---> 13 ---> 14 ---> 15 ---> 16 -->
  ---> 17 ---> 18 ---> 19 ---> 20 ---> 21 ---> 22 ---> 23 ---> 24 -->
  ---> 25 ---> 26 ---> 27 ---> 28

Module: demo_mod     Task: demo_task
Variable: wurz

Element-Nr.: 19
 a= 64
 b= 134
 ca [0]= 'A'
    [1]= '\0'
 next= 0x5a76
```

Fig. 4: Visualization of Linked Lists

5. Results and Conclusions

Features and concepts for the design and implementation of tool environments for concurrent multiprocessors were presented. In conclusion the presented tool environment MMS offers the following advantages:

* *The programmer can look interactively at the dynamic behavior of parallel computers and their program execution using several tools (debugger, performance analyzer, visualizer).*
* *All tools support various abstraction levels (machine level, C language level and process level).*
* *All tools of the environment are integrated using a unique event and action mechanism. In addition, the tools are based on the same symbol data base and they use a common graphic and menu driven human interface.*
* *The event and action mechanism is based on different instrumentation techniques (hardware, software, hybrid). A mix of instrumentation techniques is possible across different processor elements.*
* *The central design concept of MMS is a hierarchical layered model for tool environments. This layered model is the reason for the portability of the environment to various multiprocessor architectures, the expandability with new tools and the adaptability to new compilers and programming concepts.*

Although the paper mainly focuses on host/target environments, the presented hierarchical model is additionally suitable for native tool environments of multiprocessors. The apparent conclusion of this paper is that the MMS is an adequate instrument for looking at the dynamic behavior of parallel computers and their concurrent software.

References

[1] T. Bemmerl: *Realtime High Level Debugging in Host/Target Environments;* Proc. of EUROMICRO Symp. on Microarchitectures, Developments and Applications, Venice, Sept. 1986, p. 387 - 400

[2] T. Bemmerl, G. Schöder: *A Portable Realtime Multitasking Kernel for Embedded Microprocessor Systems;* Proc. of EUROMICRO Symp. on Microcomputers: Usage, Methods and Structures, Portsmouth, Sept. 1987, p. 181 - 188

[3] H. Burkhart, R. Millen: *Monitoring Tools for Multiprocessor Environments;* Proc. of Int. Conf. on Parallel Computing 85, 1985, p. 345 - 351

[4] R. Klar, N. Luttenberger: *VLSI-based Monitoring of the Inter-Process-Communication in Multi-Microcomputer Systems with Shared Memory;* Proc. of EUROMICRO' 86, Venice, Sept. 1986

[5] J. Nievergelt, B. Plattner: *Monitoring Program Execution: A Survey;* IEEE Computer, Nov. 1981

[6] T.W. Pratt: *The PISCES 2 Parallel Programming Environment;* IEEE Int. Conf. on Parallel Processing, 1987, p. 439 - 445

[7] Z. Segall, L. Rudolph: *PIE: A Programming and Instrumentation Environment for Parallel Processing;* IEEE Software, Nov. 1985, p. 22 - 37

[8] R. Seidner, N. Tindall: *Interactive Debug Requirements;* ACM Symp. on High Level Debugging, Pacific Grove, March 1983, p. 9 - 22

[9] L. Snyder, D. Socha: *Poker on the Cosmic Cube: The First Retargetable Parallel Programming Language and Environment;* IEEE Int. Conf. on Parallel Processing, 1986, p. 628 - 635

Viewing Anomalous States in Parallel Programs

Charles E. McDowell

Computer and Information Sciences
University of California, Santa Cruz
Santa Cruz, California 95064

ABSTRACT

Static analysis techniques have been developed for detecting anomalies in parallel scientific applications programs. The analysis is based on the generation of a state graph called the Concurrency History Graph. This graph contains all possible parallel states of the program. Displaying the Concurrency History Graph on a large screen multi-window workstation can greatly enhance the understanding of the anomalies reported and the program in general. This is an important aid to debugging and understanding parallel programs.

1. Introduction

Any attempt to observe or control the internal execution of a non-deterministic program may result in a change in the behavior and resulting output of the program. This has been referred to as the *probe effect* or the Hiesenberg uncertainty principle applied to parallel programs. The probe effect makes the use of conventional dynamic debugging techniques based on breakpoints, tracing and single stepping ineffective for isolating many bugs in parallel programs. In these situations an important alternative is the use of static analysis which can detect certain classes of anomalies in parallel programs. An anomaly is a potential error. An anomaly may not be an actual error because the execution path containing the anomaly may be infeasible[1], or because the programmer intended the program to exhibit behavior in certain situations that is normally indicative of an error.

In ART[AM85] static analysis is being used to detect two classes of errors in parallel programs: *synchronization* errors and *data-usage* errors. Deadlock is a familiar type of synchronization error. Data-usage errors include the usual sequential data-usage errors such as reading an uninitialized variable, and parallel data-usage errors typified by two processes simultaneously updating a shared variable.

In ART, the static analysis is based on the creation of a *concurrency history graph* (CHG) similar to that described by Taylor.[Tay83] Nodes in the CHG correspond to possible states of the parallel program, and edges correspond to one or more tasks in the parallel program advancing from one synchronization point to another. From this graph it is possible to derive some important anomalies. The size of the CHG can be quite large. In general it may grow in size exponentially with respect to the number of parallel tasks. Techniques have been developed to keep the size of the CHG manageable, although still exponential in the worst case.

The result of applying static analysis is an anomaly report. The presentation of the anomaly report should provide the user with sufficient information to determine easily if the anomaly is actually an error, and if so, how the error can be repaired. In order to eliminate an error due to improper synchronization of a parallel program, the user must be able to determine how the erroneous concurrent state could arise. For example, it may not be sufficient to report that variable X is modified concurrently by a process executing line 100 and another process executing line 200. If the user cannot understand how 100 and 200 could be executing in parallel, then it may be difficult to determine how to resolve the problem. Furthermore, the user may simply decide (erroneously) that this situation could never arise and that the anomaly report should be ignored.

The approach taken in ART is to allow the user to examine the concurrency state resulting in the anomaly report and also to examine the concurrency states that led up to that state. The *appearance* is similar to that found in some dynamic debuggers for parallel programs.[Gri87, Seq86] However, the dynamic debuggers require the program to be executed and suffer from the probe effect. In contrast, ART does not execute the program, thus avoiding the probe effect. Before describing the user interface and presentation of the anomaly report, a brief description of the CHG is presented in the next section.

2. The Concurrency History Graph

The analysis is based upon the program *control flowgraph*, in which each node represents a sequence of 'straight-line' code terminated by a transfer of control, or a synchronization operation. Edges in the control flowgraph represent sequential and branch transfers of control, and task creation. The *synchronization graph* is a compressed version of the control flowgraph. All nodes in the control flowgraph can be classified as either sequential or synchronization. A node is a synchronization node if it represents a synchronization operation (e.g. wait, create, etc.) or represents a function or procedure invocation where the flowgraph of the invoked procedure contains one or more synchronization nodes. A *sequential path* in the control flowgraph is a path in which all nodes are sequential. The synchronization graph is derived from the control flowgraph by eliminating all sequential nodes and connecting the synchronization nodes with edges corresponding to the sequential paths in the control flowgraph. The synchronization graph for the example source in the Appendix is shown in Figure 1.

1. The detection of infeasible paths has been shown to be equivalent to the halting problem.[Cla76]

Two task states can be associated with each synchronization operation. These two states are referred to as *pre-* and *post-* transition states. In a pre-transition state, a task is waiting to perform a synchronization operation. In a post-transition state, a task has just completed a synchronization operation. Thus, the synchronization operation actually occurs during the transition from a pre- to a post-transition state. A *complete transition* is a sequence of non-synchronization operations beginning immediately after a synchronization operation up to and including exactly one successor synchronization operation. A complete transition is therefore, a transition from one post-transition state to another post-transition state. A task *advances* by making a complete transition.

The central data structure used in our algorithm is the *concurrency history graph*, CHG. The nodes of a CHG are *concurrent states*, and the edges are *history transitions*. A concurrent state corresponds to a low resolution snapshot of the execution of a parallel program. The only variables that are ''captured'' are those used in synchronization operations, and the ''program counter'' of each task is only resolved to the nearest preceding synchronization operation. Windows 2-6 in Figure 2 are displaying a single concurrency state. Every possible distinct concurrent state of a program is represented in the CHG for the program (given the restrictions on resolution). However, many distinct concurrent states may be represented by the same node in the CHG (see[McD88] for details). Each edge corresponds to one or more tasks making a complete transition, which is the minimum change necessary to generate a distinct concurrency state. Only *complete transitions* are explicitly represented in the CHG (i.e. all tasks are in post-transition states).

Given a particular initial concurrency state, the CHG is generated by advancing tasks from one state to another whenever possible, creating new concurrency states. The CHG edges are labelled with sets that represent all non-synchronization variables that are read or written by the corresponding complete state transition. These sets are called read-write sets. The synchronization variables are those variables accessed atomically by the synchronization operations and therefore, cannot be the source of parallel access anomalies.

Two important types of anomalies can be directly deduced from the CHG. The first contains both deadlock and wait forever which are collectively referred to here as *synchronization anomalies*. These are detected by the existence of a concurrency state containing non-terminated tasks[2] that cannot advance to a new state. The example in the Appendix contains a simple example of a wait forever anomaly. The main routine in the example is waiting on an event (ALL_DONE) that is never posted, resulting in a concurrency state containing only the main task that has not terminated and also cannot advance.

2. Tasks are eliminated from the concurrency state when they terminate, therefore any task in the concurrency state is non-terminated.

The second type of anomaly occurs when two concurrently executing tasks attempt to access shared data and at least one task is attempting a modify operation. This is called a *parallel access anomaly* (see Figure 2). It introduces a race condition in which the value read or the final value written into the shared data depends on the speed of execution of the tasks. A parallel access anomaly exists in a CHG if it contains a concurrency state with two tasks such that the intersection of the read-write sets of the complete transitions contains anything other than read-read intersections.

3. Viewing the CHG

The user interface supported by ART serves two purposes. First, it clearly indicates to the user, in terms of the original source, where any detected anomalies are located. In addition, it can aid in understanding how a parallel program may execute. A sample user display is shown in Figure 2. The complete program text is in Appendix 1. This program is incorrect by design for illustrative purposes. Window 6 in Figure 2 contains a text listing of the anomalies detected in the program. Anomaly number 1 is currently displayed and indicates a read/write parallel access conflict to variable MAX in CHG node number 10. It states further that the problem occurs between a task executing between lines 29-40 and a task executing between lines 35-37. The concurrency state corresponding to CHG node 10 is displayed in Windows 1-5. Window 5 contains the values of various global synchronization variables. Windows 2 and 3 display the state of the tasks associated with the anomaly. The top half of windows 1-4 is scrollable and by default displays the text around the synchronization operation corresponding to the task state. The task state is the last synchronization operation executed by the task. The highlighted line indicates the task state. The lower half of Windows 1-4 displays the next possible synchronization operation to be performed by the tasks represented by the windows. In Window 3 there are two possible successors, however, because the lock is already set by the task in Window 2, the task in Window 3 could actually only advance to the new state at line 40.

Windows 7 and 8 contain the synchronization graph and the concurrency history graph. Only a portion of the graphs is displayed centered on the nodes corresponding to the concurrency state displayed in windows 1-5. These windows can be expanded to fill the entire screen if desired, displaying the complete graphs.

The remaining window is the control window. ''Back'' displays the CHG node displayed previously. ''Next'' displays the sequentially next CHG node. ''Plot CHG'' expands the CHG graph display to fill the entire display. If the graph still does not fit on the display, the window can be scrolled both horizontally and vertically. ''Plot SAF'' does the same expansion for the program graph (Synchronization Augmented Flowgraph). ''Pick#'' prompts for a CHG node number and displays that node. This is the main way of moving around when viewing anomalies, since the anomalies are reported in terms of

CHG node numbers. "Quit" is self explanatory.

There is one additional -- the most useful -- method for moving about in the CHG. By selecting one of the successor states in the lower half of windows 1-4 with the mouse, a CHG node will be displayed (if one exists) with at least one task from the current window[3] advanced to the selected synchronization operation. This allows the user to browse the concurrency states in a manner analogous to course grained single stepping.

4. Conclusion

A system for browsing the possible concurrent states of a parallel program was presented. It is part of an anomaly reporting tool currently used as an aid for debugging programs written in Fortran with extensions for explicit parallelism.

Work is in progress to integrate this static analysis tool with a dynamic debugger. Taylor[Tay84] describes several ways in which static analysis could be productively combined with dynamic analysis. A primary goal is to use paths through the concurrency history graph to force deterministic executions of a parallel program under control of the dynamic debugger. Conversely, information from dynamic monitoring will be used to guide partial static analysis when complete static analysis would generate too many states.

Appendix 1.

This program intentionally contains errors. It is intended to normalize an array by the maximum value in the array using N=4 parallel tasks.

```
      PROGRAM EXAMPLE

      parameter(N=4)
      COMMON CMAX(10)
      COMMON /M2/ MAX
      DIMENSION A(10,10)
      _declare_barrier(B1)
      _decl_common_do(COLMAX)
      _decl_common_do(NRML)
          _decl_event(ALL_DONE)

      _init_barrier(B1,N)
      _init_common_do(COLMAX,1,N)
      _init_common_do(NRML,1,N)

      _create_family(TASK_ID,NORMAL,A,N)
      _wait_event(ALL_DONE)
      K=1
      END

      _task_entry(NORMAL,A,N)
      _decl_common_do(COLMAX)
      _decl_common_do(NRML)
      _declare_barrier(B1)
      COMMON CMAX(10)
      COMMON /M2/ MAX
      DIMENSION A(10,10)

      _begin_common_do(colmax,J)
       DO 100 I=1,N
         IF(CMAX(J) .LT. A(I,J)) CMAX(J) = A(I,J)
    100 CONTINUE
       IF(CMAX(J).LT.MAX)GOTO 150
        _lock_on(MAXLOCK)
        MAX = CMAX(J)
        _lock_off(MAXLOCK)

    150 CONTINUE
      _end_common_do(colmax,J)

      _wait_barrier(B1)

      _begin_common_do(NRML,J)
        DO 200 I=1,N
        A(I,J) = A(I,J) / MAX
    200 CONTINUE
      _end_common_do(NRML,J)

      _task_return
      END
```

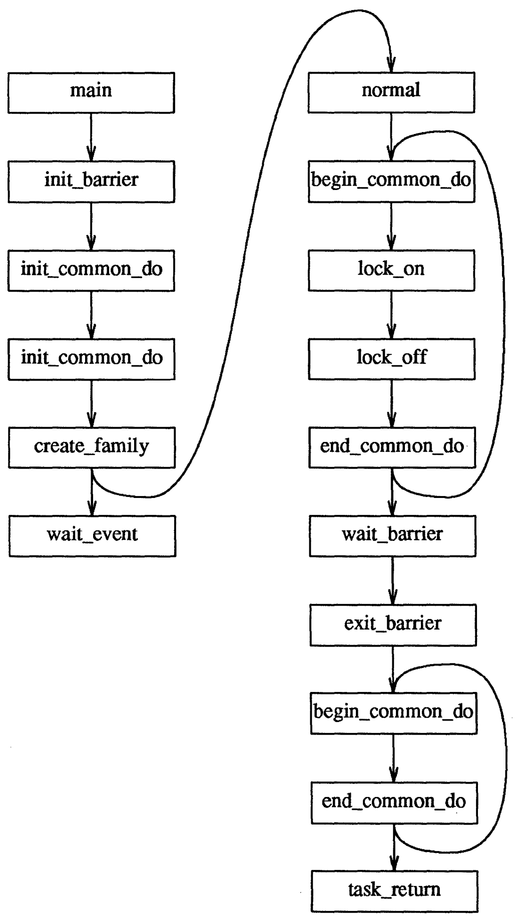


Figure 1. Synchronization Graph.

3. Each window may represent more than one task all in the same state. This collection of tasks is called a family. In addition families are sometimes grouped into clans. This is done to reduce the size of the CHG. Details on the use of clans and families to reduce the size of the CHG can be found in another report.[McD88]

References

AM85.
Appelbe, W. F. and McDowell, C. E., ''Anomaly Reporting - A Tool for Debugging and Developing Parallel Numerical Algorithms,'' *Proc. First Inter. Conf. on Supercomputing Systems*, pp. 386-391, December 1985.

Cla76.
Clarke, L.A., ''A System to Generate Test Data and Symbolically Execute Programs,'' *IEEE Transactions Software Engineering*, vol. SE-2, no. 3, pp. 215-222, September 1976.

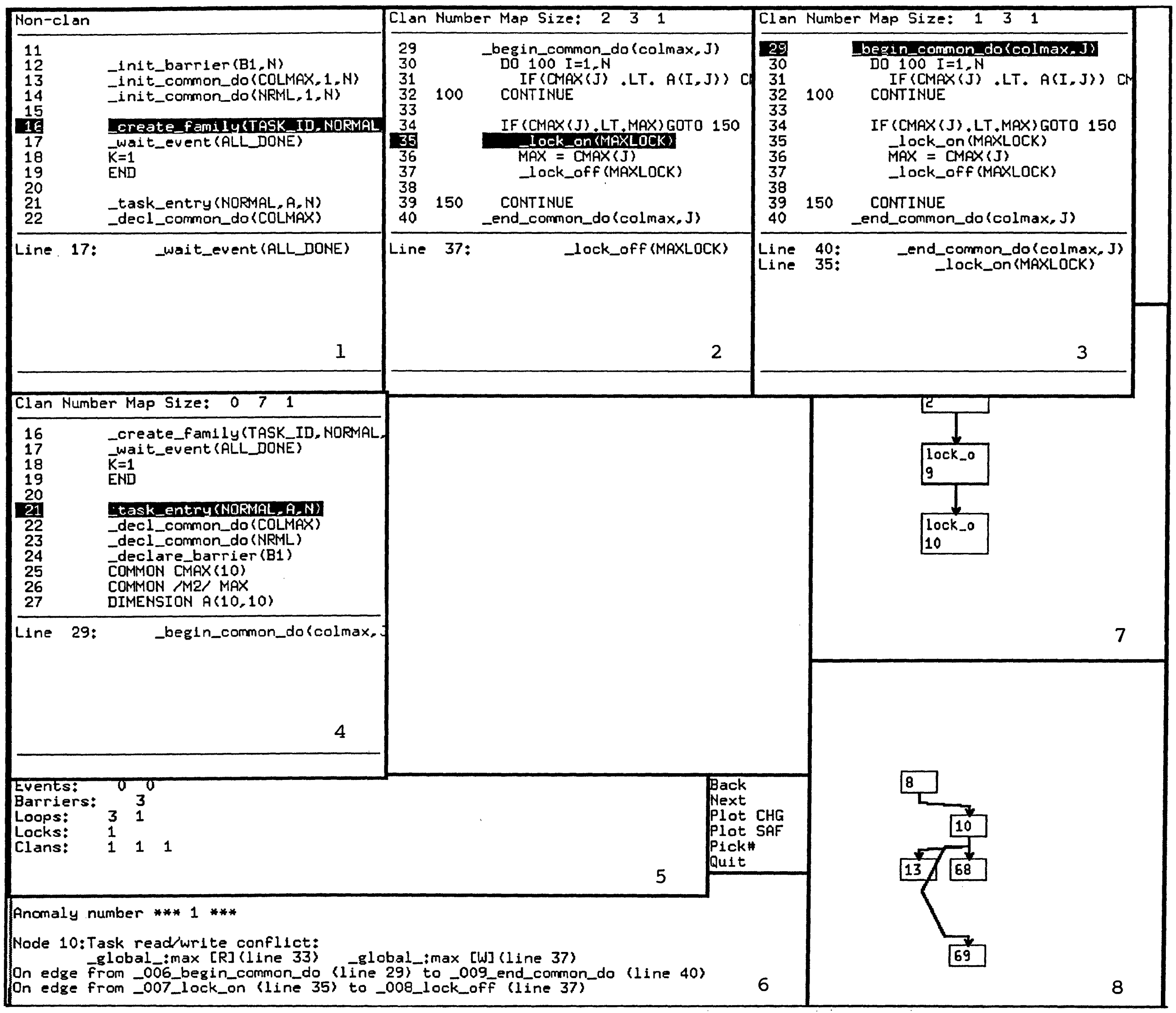
Gri87.Griffin, J., ''Parallel Debugging System User's Guide,'' *LANL Tech Report*, July 1987.

McD88.
McDowell, C. E., ''A Practical Algorithm for Static Analysis of Parallel Programs,'' *Journal of Parallel and Distributed Computing*, to appear 1988.

Seq86.
Dynix Pdbx Parallel Debugger User's Manual, Sequent Corp., 1986.

Tay83.
Taylor, R. N., ''A General-Purpose Algorithm for Analyzing Concurrent Programs,'' *CACM*, vol. 26, no. 5, pp. 362-376, May 1983.

Tay84.
Taylor, R. N., "Debugging Real-Time Software in a Host-Target Environment," *U.C. Irvine Tech. Rep. 212*, 1984.



```
Non-clan

 11
 12        _init_barrier(B1,N)
 13        _init_common_do(COLMAX,1,N)
 14        _init_common_do(NRML,1,N)
 15
 16        _create_family(TASK_ID,NORMAL
 17        _wait_event(ALL_DONE)
 18        K=1
 19        END
 20
 21        _task_entry(NORMAL,A,N)
 22        _decl_common_do(COLMAX)

Line 17:        _wait_event(ALL_DONE)
                                                 1
```

```
Clan Number Map Size:  2  3  1

 29         _begin_common_do(colmax,J)
 30           DO 100 I=1,N
 31             IF(CMAX(J) .LT. A(I,J)) C
 32   100     CONTINUE
 33
 34           IF(CMAX(J).LT.MAX)GOTO 150
 35             _lock_on(MAXLOCK)
 36             MAX = CMAX(J)
 37             _lock_off(MAXLOCK)
 38
 39   150     CONTINUE
 40         _end_common_do(colmax,J)

Line  37:            _lock_off(MAXLOCK)
                                                 2
```

```
Clan Number Map Size:  1  3  1

 29         _begin_common_do(colmax,J)
 30           DO 100 I=1,N
 31             IF(CMAX(J) .LT. A(I,J)) CM
 32   100     CONTINUE
 33
 34           IF(CMAX(J).LT.MAX)GOTO 150
 35             _lock_on(MAXLOCK)
 36             MAX = CMAX(J)
 37             _lock_off(MAXLOCK)
 38
 39   150     CONTINUE
 40         _end_common_do(colmax,J)

Line  40:        _end_common_do(colmax,J)
Line  35:            _lock_on(MAXLOCK)
                                                 3
```

```
Clan Number Map Size:  0  7  1

 16        _create_family(TASK_ID,NORMAL,
 17        _wait_event(ALL_DONE)
 18        K=1
 19        END
 20
 21        _task_entry(NORMAL,A,N)
 22        _decl_common_do(COLMAX)
 23        _decl_common_do(NRML)
 24        _declare_barrier(B1)
 25        COMMON CMAX(10)
 26        COMMON /M2/ MAX
 27        DIMENSION A(10,10)

Line  29:       _begin_common_do(colmax,J
                                                 4
```

```
Events:     0  0
Barriers:      3
Loops:     3  1
Locks:     1
Clans:     1  1  1
                                                 5
```

```
Anomaly number *** 1 ***

Node 10:Task read/write conflict:
        _global_:max [R](line 33)    _global_:max [W](line 37)
On edge from _006_begin_common_do (line 29) to _009_end_common_do (line 40)
On edge from _007_lock_on (line 35) to _008_lock_off (line 37)
                                                 6
```

Figure 2. User Display with Parallel Access Anomaly.

PAT--An Interactive Fortran Parallelizing Assistant Tool

Kevin Smith
William F. Appelbe
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

Abstract -- The development of parallel programs has inherent difficulties which demand new tools to help develop reliable and efficient parallel programs. This paper describes one such tool, an interactive Fortran Parallelizing Assistant Tool (*Pat*), as part of a multitasking toolkit under development which also contains tools for static analysis and dynamic debugging of parallel programs.

The parallelization tool is designed to assist and educate users in converting sequential programs into parallel programs; it relies on dependence analysis and programmer interaction in introducing and modifying parallel constructs. Significant differences from other work in this area are the tool's ability to recognize parallel constructs and handle partially parallel programs, the ability to parallelize loops with embedded critical sections, and the adaptability and portability of the tool.

Introduction

The advantages of today's parallel computing capabilities are often lost due to the complexities of parallel programming. Constructs used to control parallelism are often difficult to learn and understand, and inherent in parallel execution are a new class of problems such as race conditions which do not reveal themselves to normal sequential debuggers. One solution is automatic parallelization such as that done by some optimizing compilers, but automatic detection of 'large grain' parallelism is extremely difficult and can only convert deterministic sequential algorithms to equivalent deterministic parallel algorithms[3].

Our approach is based upon a toolkit which will help a programmer master the complexity of parallel programs by assisting in parallelizing sequential or partially parallel programs and in debugging and testing multitasking programs. This toolkit consists of:

- A *parallelizer* which will examine a source program and suggest parallelizing modifications.
- A *static analyzer* which will simulate the execution of a source program to locate anomalies caused by the interaction of the tasks.
- A *dynamic debugger* which will interactively execute a source program.

The static analysis tool has been implemented and is in use, and a prototype of the parallelizer has been implemented at present.

The Toolkit

The parallelizing tool is built on the work described in [3] and referenced in [6], and uses the same parallel primitives and underlying structure as *Art* (Anomaly Reporting Tool), the prototype static analyzer portion of the toolkit.

The parallel model assumed is the SPMD model of shared memory multitasking; this treats a program as a single thread of control which creates other tasks as needed to execute designated sections of code in parallel. The basic parallelization technique used is execution of sequential program loop iterations with separate parallel tasks, and use of critical sections and pipelines to protect parallel access anomalies and dependences between iterations.

Art performs an exhaustive static analysis of a parallel program by constructing the complete concurrency history[9] of the program; it detects three classes of potential bugs, or anomalies. *Non-deterministic variable* anomalies are references to variables that depend upon task scheduling. *Synchronization* anomalies include deadlock and busy-waiting loops. *Parallel access* anomalies occur when a task tries to *write* to a variable which another task is trying to access simultaneously; the scheduling of tasks will determine which will win this 'race' condition.

Presently the tools in the toolkit recognize

This work was supported by DOE contract number W-7405-Eng-36 through Los Alamos National Laboratory.

a set of parallel primitives described in [4], similar to those of Cray microtasking, although Pat is designed to be retargetable to other shared memory multitasking primitives, such as the 'Fork-Join' model.

The Parallelizing Tool

Much of our work on *Pat* parallels Rice University's work with *PTOOL* which interactively displays dependence analysis [1]; however, our approach differs from theirs primarily in that:

- *Pat* suggests actual program modifications and modifies source code in addition to displaying dependences.
- *Pat* recognizes parallel primitives and handles partially parallel programs with the capability of suggesting modifications to existing parallel structures.
- 'Large grain loops' which have embedded critical sections are parallelized.
- *Pat*, based on a modified 4.2 BSD Unix Fortran77 compiler, is portable, using an adaptable front-end which can accept various primitives, and X windows to display source code and program dependences graphically (this is illustrated for a small program in Diagram 1).

The strength of our approach lies in the mixed-initiative approach of interaction between programmer and tool. There are two phases to the parallelization process: a dependence analysis phase, and the interactive parallelization phase.

Dependence Analysis

The first step in parallelization is the extraction of data dependence information; a description of dependences and their usefulness can be found in [2]. Essentially, it is necessary to detect any variable reference which might be affected if a *write* of that variable were performed in non-sequentially. *Pat* employs *Art*'s "best estimate" strategy of identifying potential dependences.

The program analysis is built on a *control flow graph* (CFG) of the program. Each node in the CFG represents either a basic block of the program or a branch or merging in the program flow[10]. Subroutine calls in the CFG are expanded inline for simplicity (i.e. each call is analyzed individually in context). Inline expansion has a low overhead in practice with a non-recursive language such as Fortran, aids in tracking aliased variables through subroutine calls, and provides more options for optimization.[a]

The desired dependence information is extracted by tracing paths through the program, using reference lists to construct a global *dependence graph*. Three classes of dependence are reported: a *read* to a preceding *write* (true dependence), a *write* to a preceding *read* (anti dependence), and a *write* to a preceding *write* (output dependence).

Pat also identifies dependences arising from concurrent operations. Each access to global data in a parallel code section is dependent on each *write* of that data in concurrent sections. In detecting parallel dependences *Pat* ignores references protected by a *lock*, and performs *subscript analysis*. Subscripts composed of expressions yielding a different fixed value or range of values will never conflict, nor will subscripts which reference the same variables in such a way that the variable value will always differ. Particularly, subscripts can vary directly with a parallel loop control variable, so that subscripts from two different iterations will not conflict.

Browsing of dependences is provided through an interactive graphic interface. Dependences can be viewed sequentially, or selectively by line number or variable name (see W1, W2 and W4 in Diagram 1).

Parallelization

The major focus of this research is the conversion of sequential loops to parallel loops. *Pat* offers a choice of loops to be parallelized. Three parallelization operations are provided: modifying parallel primitives in an existing task, converting a sequential loop to parallel execution, or designating segments of sequential code to be run as individual tasks.

A special case of loop construction is an 'if loop'. Cycles in the CFG represent 'if loops' in the program which in some cases are equivalent

[a] *Pat* treats each call to a procedure as a separate code section for the purposes of parallelization analysis, forcing the user to ensure that parallelization modifications for one call do not conflict with those for another call. This could require duplication of the procedure body. *Pat* provides assistance in this sort of checking.

to 'do loops' and can be executed in parallel. If the variable to be used as the loop control variable is in the form of an *induction variable* (see below), it can be extracted automatically. Otherwise, the programmer can interactively provide this information.

Locating large blocks of non-loop code which might be executed concurrently is more difficult. Tightly coupled segments of sequential code which have few or no dependences on surrounding code can be shifted to a private task. The programmer can suggest areas to the tool and it will help parallelize them by displaying dependences in and surrounding the code section.

There will still be poorly structured programs which are unanalyzable by such a tool [5]. In such situations *Pat* shows dependences and suggests alternative better structured primitives; it is then up to the programmer to restructure the program.

Loop Analysis

After the user selects a loop body, Pat determines which variables referenced inside the loop can be made *local* to tasks executing the loop body and which will best remain global or *shared*. A subset of *shared* variables must also be *ordered* by using *events* to ensure sequencing of assignments to global data. Currently *Pat* distinguishes *local*, *shared*, *shared locked*, and *shared ordered* variables as in [7] (see W3 in Diagram 1).

Recognition of *induction variables* is also useful. Induction variables are those with a single assignment of the form <variable> = <variable> <op> <expression>. Unmodified, they are global variables requiring ordering, but they can be replaced with a local variable assigned to a function of the loop control variable.

Pat must also decide whether or not a *barrier* will be required at the end of the parallel loop; this will be so if any values generated in the loop are read later. For example, if the loop is performing a summation of an array, a local sum can be employed in each task and the global sum increased by each locally calculated value (see the variable 'sum' in the sample program in Diagram 1). A barrier is required to assure that all tasks add their local sum to the final value before the code following the loop is allowed to proceed.

Guarding Dependences

Once the parallel region is determined parallel dependences in that region must be dealt with. Each dependence in the code section is identified, with a list of possible transformations and a suggestion from *Pat* as to how this dependence is to be protected; the programmer can accept the recommendation, protect the dependence differently, or ignore it. The programmer can also query the tool as to why it selected a particular modification or reported a particular dependence (see W4 in Diagram 1).

Several code modifications are effective in avoiding the necessity of synchronization guarding between iterations of a loop; these include *subscript alignment, code replication, code shifting* and *node splitting*. Each of these involves alterations in the loop body to minimize or avoid access of conflicting variables. *Pat* identifies the possibility of code modifications, suggests those best fitting the program environment, and indicates to which lines of the code they should be applied.

In cases where the complexity is too great for these modifications, or in which the user does not wish to modify the code accordingly, parallel primitives must be added to explicitly protect the dependent references. *Pat* guides in inserting *locks* or *events* to protect these.

Future Work

The following parallelization optimization expansions are being added to *Pat*. Some of these transformations are in opposition to each other, requiring care that application of different techniques does not counteract others. Specific transformations are described in more detail in [8] and [2].

One concern once a candidate area is selected for parallelization is optimizing the scope of the parallel area. For example, *Pat* can help decide the portion of a task which must be enclosed in a parallel do. Lines which are not required for each iteration, but are required for execution of the loop, can be interactively shifted outside the parallel do area.

Modification of the structure of loops can be made easier by *loop normalization*; this is another conversion which can be done internally. As the loop control variable often controls expression values, this also helps in expression comparison, as for array subscript analysis, by reducing functions dependent on the loop control variable to a similar form.

There are a group of modifications aimed at enhancing the function of a specific computer. These include *loop interchange, loop fusion* or *fission, loop collapsing*, and again *code shifting*. They work on the structure of the loop and the code surrounding it to optimize it for a specific purpose such as matching tasks to a specific number of processors, or fitting array size to the size of a cache memory.

These optimizations depend on specific knowledge of the target machine. In some cases it is advantageous to divide loops to give a greater number of small tasks, and in others it is better to merge other code into the task area to give a larger body for a few tasks. For example, if a machine runs 4 processors, 12 smaller tasks would be more likely to distribute evenly than 6 large tasks. Such "strip mining" can allow optimal employment of a machine's capabilities when target specifics are known.

Status

Currently, the parallelizing tool performs the following actions:

- extraction of variable references and construction of dependence graph;
- browsing of dependences;
- identification of parallel tasks, do-loops and if-loops for parallelization;
- automatic insertion of parallel primitives for task creation;
- identification of parallel dependences;
- interactive insertion of necessary parallel protections for variables.

The parallelizing tool and the integrated static analyzer were developed from an F77 compiler front end, operating under 4.2 BSD Unix. The tools use the X window system for graphic output. They have been ported to Sequent, SUN, VAX and ISI workstations.

Conclusion

The need for tools to assist in dealing with parallelism is clear. Our goal is a toolkit which will provide help in debugging and correcting parallel code, and will assist both in modifying a sequential program to use parallel constructs and in attaining the optimum in parallelism and clarity while teaching clear parallel programming.

The static analyzer is the first part of the debugging portion of the toolkit; it examines parallel code in detail and pinpoints a number of potential errors unique to parallelism. The dynamic debugger will complement this analysis by allowing the user to follow the parallel execution of a program.

The parallelizing assistant tool extracts dependence information from an analysis of the source code, and then guides the user in adding efficient parallel structures to the program. It recognizes parallel constructs and suggests modifications to run iterations of loops as parallel loops, as well as indicating synchronization and variable protection which will be required by the parallelized loop.

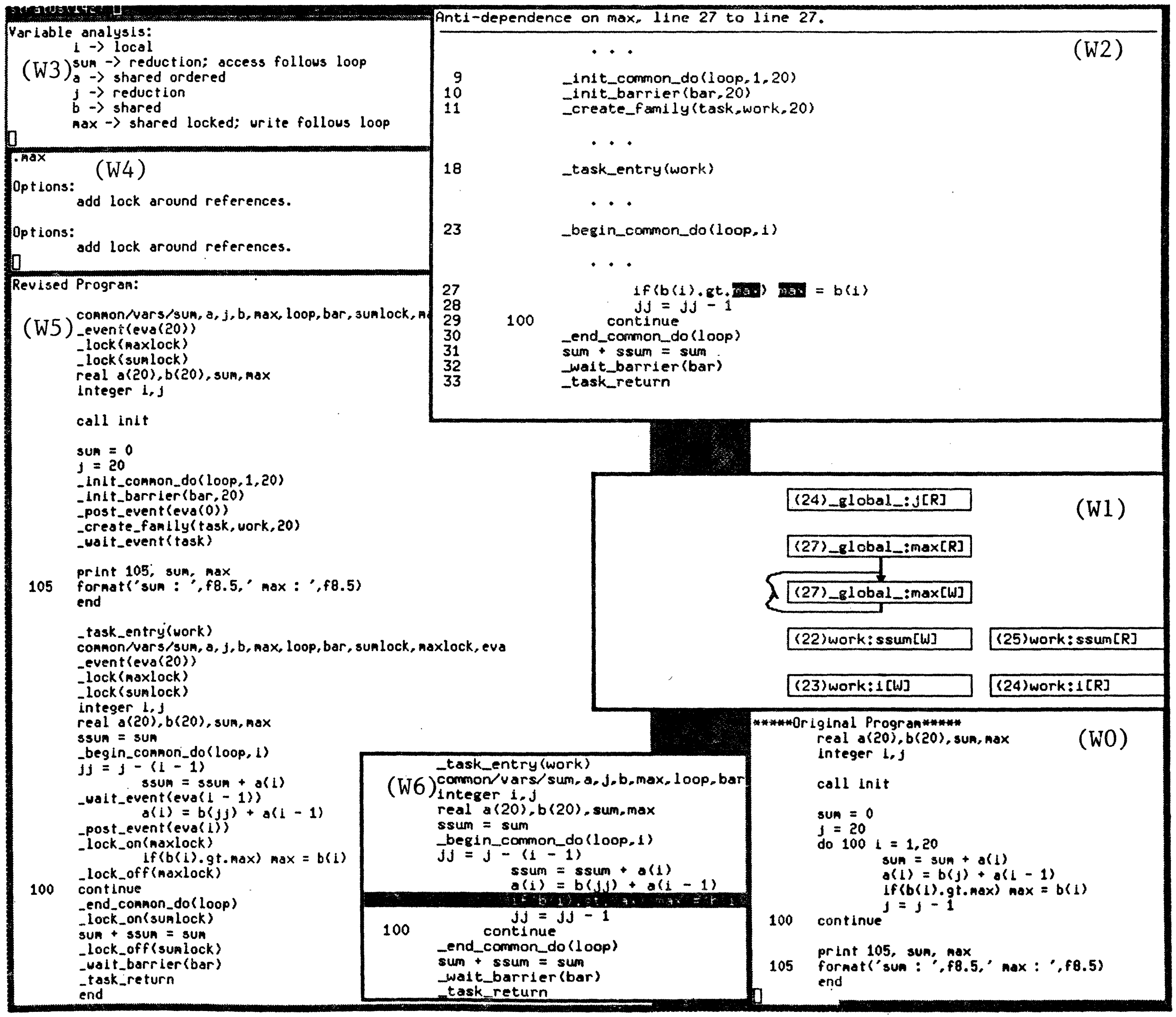
References

1. Randy Allen, Donn Baumgartner, Ken Kennedy, and Allan Porterfield, "PTOOL: A Semi-automatic Parallel Programming Assistant," Computer Science Technical Report, Rice University (January 1987).
2. Randy Allen and Ken Kennedy, "Automatic Translation of Fortran Programs to Vector Form," Computer Science Technical Report, Rice COMP TR84-9, Rice University (July, 1984).
3. Bill Appelbe and Charles E. McDowell, "Developing Multitasking Applications Programs," Proceedings of the Hawaii International Converence on System Sciences (January 1988).
4. Brian F. Hanks and Charles E. McDowell, "A Proposed Tool for Parallel Programming Multiple 3081/E's," Computer Science Technical Report UCSC-CRL-86-20, University of California at Santa Cruz (July 25, 1986).
5. Leslie A. Henderson, *The Usefulness of Dependency-Analysis Tools in Parallel Programming: Experiences Using Ptool,* Los Alamos National Laboratory (1988).
6. Charles E. McDowell, *A Formal Model for Static Analysis of Parallel Programs,* University of California, Santa Cruz (1987). to appear in Journal of Parallel and Distributed Computing.
7. Anita Osterhaug, *Guide to Parallel Programming on Sequent Computer Systems,* Sequent Computer Systems, Inc., Beaverton, Oregon (1986).

8. David A. Padua and Michael J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM* **29**(12), pp. 1184-1201 (December 1986).

9. Richard N. Taylor, "A General-purpose Algorithm for Analyzing Concurrent Programs," *CACM* **26**(5), pp. 362-376 (May 1983).

10. William W. Waite and Gerhard Goos, *Compiler Construction,* Springer-Verlag, New York (1985).



```
Variable analysis:
        i -> local
        sum -> reduction; access follows loop
        a -> shared ordered
        j -> reduction
        b -> shared
        max -> shared locked; write follows loop
```

```
.max
Options:
        add lock around references.

Options:
        add lock around references.
```

```
Revised Program:

        common/vars/sum,a,j,b,max,loop,bar,sumlock,m
        _event(eva(20))
        _lock(maxlock)
        _lock(sumlock)
        real a(20),b(20),sum,max
        integer i,j

        call init

        sum = 0
        j = 20
        _init_common_do(loop,1,20)
        _init_barrier(bar,20)
        _post_event(eva(0))
        _create_family(task,work,20)
        _wait_event(task)

        print 105, sum, max
 105    format('sum : ',f8.5,' max : ',f8.5)
        end

        _task_entry(work)
        common/vars/sum,a,j,b,max,loop,bar,sumlock,maxlock,eva
        _event(eva(20))
        _lock(maxlock)
        _lock(sumlock)
        integer i,j
        real a(20),b(20),sum,max
        ssum = sum
        _begin_common_do(loop,i)
        jj = j - (i - 1)
                ssum = ssum + a(i)
        _wait_event(eva(i - 1))
                a(i) = b(jj) + a(i - 1)
        _post_event(eva(i))
        _lock_on(maxlock)
                if(b(i).gt.max) max = b(i)
        _lock_off(maxlock)
 100    continue
        _end_common_do(loop)
        _lock_on(sumlock)
        sum + ssum = sum
        _lock_off(sumlock)
        _wait_barrier(bar)
        _task_return
        end
```

```
Anti-dependence on max, line 27 to line 27.

                 . . .
 9          _init_common_do(loop,1,20)
10          _init_barrier(bar,20)
11          _create_family(task,work,20)

                 . . .

18          _task_entry(work)

                 . . .

23          _begin_common_do(loop,i)

                 . . .

27                  if(b(i).gt.max) max = b(i)
28                  jj = jj - 1
29      100      continue
30          _end_common_do(loop)
31          sum + ssum = sum
32          _wait_barrier(bar)
33          _task_return
```

```
        _task_entry(work)
        common/vars/sum,a,j,b,max,loop,bar
        integer i,j
        real a(20),b(20),sum,max
        ssum = sum
        _begin_common_do(loop,i)
        jj = j - (i - 1)
                ssum = ssum + a(i)
                a(i) = b(jj) + a(i - 1)
[illegible]
                jj = jj - 1
 100         continue
        _end_common_do(loop)
        sum + ssum = sum
        _wait_barrier(bar)
        _task_return
```

```
*****Original Program*****
        real a(20),b(20),sum,max
        integer i,j

        call init

        sum = 0
        j = 20
        do 100 i = 1,20
                sum = sum + a(i)
                a(i) = b(j) + a(i - 1)
                if(b(i).gt.max) max = b(i)
                j = j - 1
 100    continue

        print 105, sum, max
 105    format('sum : ',f8.5,' max : ',f8.5)
        end
```

Diagram 1. Screen Display
Windows can be moved, resized or scrolled.

W0: Original sequential program.	W4: Control window.
W1: Part of dependence graph.	W5: Parallel program created using *Pat*. (parallel primitives are as in [4])
W2: View of a single dependence.	W6: Intermediate task code for a parallel loop.
W3: Loop variable analysis.	

Heuristic Rule-Based Program Transformations for Enhanced Vectorization

Pradip Bose

IBM Research Division
T. J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598

Abstract: An example of a state-of-the-art high-end mainframe is the IBM 3090VF; its associated vectorizing compiler, the IBM VS FORTRAN (version 2) compiler, incorporates some of the latest techniques in automatic vectorization and code optimization. Advances in hardware and compiler technology not withstanding, a potential limitation is the "knowledge gap" which exists between the average end user and the compiler/machine sub-system. In particular, the user often does not know how to write source code which will result in generation of efficient, high performance object code. In this short paper, we present an overview of a research project, called EAVE, which is a knowledged-based approach for bridging the knowledge gap alluded to above.

1.0 Introduction

The IBM VS FORTRAN (version 2) compiler[7] is an example of a state-of-the-art vectorizing compiler. It generates vectorized code for execution on the IBM 3090 with vector facility (VF)[3-4], which is IBM's latest high-end mainframe. The compiler has the advantage that source to source transformations are transparent to the users; vectorization is viewed almost as an additional level of optimization, which the user can invoke as an option. Such advances in compiler technology not withstanding, a potential limitation is the *knowledge gap* which exists between the average end user and the compiler/machine sub-system. As a result of this gap, users frequently write high-level source code which achieves less than achievable performance. Although the VS FORTRAN (version 2) programming guide[1] provides some pointers for writing vectorizable code, serious users have felt the need to acquire greater expertise in fully exploiting the capabilities of the IBM 3090 with vector facility (VF). Some published material (e.g.,[5-7]) has proved to be useful. However, astute experimentation by several people, e.g.[8-9], has identified educated heuristics for rewriting FORTRAN code in order to achieve near-peak performance. Such ideas are not readily apparent from the earlier published material. In essence, expertise in writing (or converting to) high performance source code, with respect to the 3090 VF, seems at present to be limited to a handful of professionals who have acquired their knowledge through purposeful experimentation and investigation; not by accident as ordinary FORTRAN users.

It is in this context that the EAVE project was started. EAVE (an Expert Advisor for VEctorization), is intended to be an intelligent aid to users of VS FORTRAN version 2, interested in efficient use of the 3090 VF. The initial version of this Pascal-based system is coded around the expert system shell: Expert System Environment/VM[18], which is a recently announced program product marketed by IBM. This project is an attempt to bridge the knowledge gap alluded to earlier. Our goal, in short, is to start with existing pieces of code supplied by the user and work *interactively* with the user to come up with the best possible program to be submitted to the compiler.

2.0 Brief Functional Description

Figure 1 shows the basic, interactive environment which is envisaged in the operation of EAVE. The user has the option of consulting EAVE before or after a trial run via the actual compiler. On a given consultation session, EAVE initially suggests possible changes in coding style and structure as an aid to enhanced vectorization and/or general code improvement. Under user control, some or all of these suggested changes may be incorporated into a modified source file, which can be processed through the actual compiler/machine system for verification of performance enhancement. In principle, this process may be iterated several times for arriving at the best possible code.

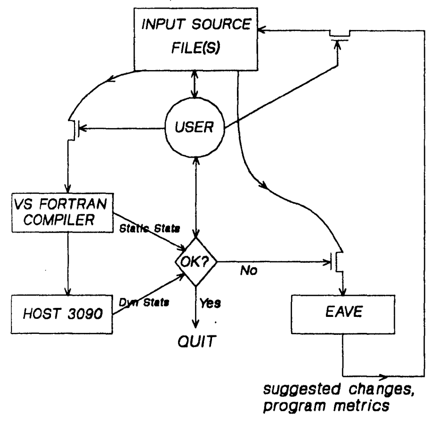


Figure 1. Interactive program development/refinement via EAVE

3.0 Heuristic Transformations and Rule Generation

In this section we illustrate our knowledge (heuristics) acquisition techniques. Certainly, the most readily accessible sources of knowledge are the innumerable papers, reports and books available on the subject. In addition to the sources already referred to[1-9], we cite a couple of others[10-12], which have been of most direct relevance to this project so far. (Additional references are, of course, available in most of the cited documents). A second source of knowledge is the *not*-so-readily available or accessible pool of actual human experts. In lieu of direct help from experts in building the knowledge base, indirect extraction of knowledge from their published papers is often the only practical alternative. The ultimate, direct approach to heuristics acquisition in the context of our very specific problem domain, is doing *appropriate experimentation* on the actual machine (IBM 3090 VF), using FORTRAN codes compiled by the actual VS FORTRAN, Version 2 compiler. The experiments must be *appropriate*, in that they must be guided by the basic knowledge available from human experts and their published expertise.

Experimentation

The basic methodology used is to compile and run FORTRAN 'do loop' kernels on the 3090 VF, under varying coding styles, and to measure relative speed-up or degradation caused by incremental source code changes. Speed (execution rate) is measured in units of millions of floating point operations per second (MFLOPS). In the following, we discuss the coding of a simple FORTRAN 'do loop' kernel, as an example in illustrating how educated experimentation can lead to derivation of logical rules for desirable program transformations. A more detailed, step-by-step explanation of rule-derivation through experimentation is available elsewhere[18]. Results based on extensive experimentation using a wide range of do loops are being published as a separate report[16].

Example Let us consider a simple matrix-vector multiplication kernel:

```
    DOUBLE PRECISION A(128, 128), B(128), C(128)
    DO 100 J = 1, N
    DO 200 I = 1, N           ...............(A)
    C(I) = C(I) + A(I,J) * B(J)
200 CONTINUE
100.CONTINUE
```

Vectorization on either I or J is possible. From stride considerations, inner loop vectorization (on I) is generally indicated. However, on actual experimentation, it is found that the decision of the compiler is to perform *outer* loop vectorization (index J)! The resulting code performs very poorly for $N \geq 80$ (see Table 1). This is an example, where the built-in economic model[7] of the compiler is effectively fooled into making a wrong decision. The economic analysis procedure is not an exact algorithm: it employs short-cuts and heuristics of its own. By analyzing the exact code patterns[16] emitted for variations of the above loop, it is possible to infer how the economic model is fooled in this case, but we shall not go into that in detail here. Essentially, beyond N = 50 or so, the large stride access begins to cause a big cache degradation.

In order to force inner loop vectorization, however, we may use the technique of inserting a temporary scalar[18]. While we are doing so, we might as well try to invoke a vector MULTIPLY-AND-ADD (VMA) compound instruction[3]. It turns out that in this context, the present version of the compiler will generate a VMA only if the scalar (B) precedes the vector. Thus, the loop structure for forcing inner loop vectorization with VMA invocation is:

```
    DO 100 J = 1, N
    DO 200 I = 1, N
    T = C(I)
    T = T + B(J) * A(I, J)   ............(B)
    C(I) = T
200 CONTINUE
100 CONTINUE
```

Note, however, that in spite of improved performance (see Table 1), there is strong reason[18] to expect even better results, if the I-loop were outermost and outer loop vectorization were performed. The best possible code, achieved after manual "outermosting", is thus:

```
    DO 100 I = 1, N
    T = C(I)
    DO 200 J = 1, N
    T = T + B(J) * A(I, J)   ...........(C)
200 CONTINUE
    C(I) = T
100 CONTINUE
```

Clearly, the reason why code (C) performs the best is that vector register reuse (via outer loop vectorization) and VMA invocation are both used in the best possible manner. In this case, each section of the vector C is loaded *once*, used for all values of J and stored back (updated) *once*; thus optimal vector register reuse is provided. The additional effect of VMA is understood, by examining the performance of code (D) below (see Table 1), in which VMA is inhibited by using the original order of multiplication:

```
    DO 100 I = 1, N
    T = C(I)
    DO 200 J = 1, N
    T = T + A(I,J) * B(J)   ...........(D)
200 CONTINUE
    C(I) = T
100 CONTINUE
```

Clearly, Table 1 demonstrates that code (C) is the best possible loop to be submitted to the compiler for this example. We are now in a position to write down the (heuristic) conditions under which a VMA-invoking transformation seems to be advisable for a high-level statement, which (on vectorization) looks like: vector1 = vector1 + scalar*vector2:

- The degree (depth) of the loop is at least 2.
- The index on which vector2 is vectorized, must belong to an outer loop, i.e., to a lower nesting level than the statement itself.

Table 1

N	Execution Rate (MFLOPS)			
	Code (A)	Code (B)	**Code (C)**	Code (D)
16	10.0	7.2	16.5	10.4
32	16.4	12.0	30.1	16.8
48	19.3	14.6	34.6	19.0
60	17.9	15.9	36.4	20.1
64	12.5	16.0	35.0	19.5
80	3.7	17.3	38.4	20.8
96	3.8	18.8	42.2	22.3
112	3.9	19.7	45.0	22.8
128	3.9	21.3	49.5	24.2

Heuristics-based loop interchange

Central to the idea of increasing vector register reuse through intelligent use of outer loop vectorization, is the ability to check for safe loop interchange. In the specific context of EAVE, this boils down to the ability of advising loop "outermosting" which will enhance performance while preserving program semantics. The theory of dependence-based loop interchange is quite well-established (e.g.[11]). This is indeed the theory which is used by the compiler[7] in testing whether a given outer loop can be "innermosted". For the purposes of our expert adviser, we seek a heuristic, "rules-of-thumb" approach for advising such loop interchange to the programmer. The basic tenet of the approach here, as in almost all the other mechanisms of heuristic transformation capability of EAVE, might be termed: *hierarchical, incremental reasoning*. Essentially, this calls for avoiding the task of solving *the most general case* first for a typical problem at hand.

Level-1 reasoning

Level-1 reasoning is based on simple pattern-directed inferencing. In the most part, such reasoning is based on *limited context* pattern matching; here, analogous to localized visual inspection by a FORTRAN expert, initial problems or characteristics are identified. Higher levels of reasoning make use of these data, if called for. In many cases, level-1 reasoning is sufficient to arrive at a decisive conclusion. The basic principle of level-1 reasoning is *looking for simplest things first.* For ease of illustration, let us limit our discussion to 2-level, perfectly nested loops. Also, let us restrict ourselves to linear subscript expressions. Let us assume that the loops are *formatted* (not *normalized*), in that all loop-headers and subscripts are converted using a uniform convention. Thus, given an initial user-supplied loop, such as:

```
    DO 100   I = 1, N
    DO  200 J = 2, N - 1
200 A(I, 1+J) = B(I) + A(I ,J)
100 CONTINUE
```

pre-formatting might convert it into something like:

```
    DO 100 I1 = 1,N,1
    DO 200 I2 = 2,N-1,1
    A(I1,I2+1) = B(I1) + A(I1,I2)
200 CONTINUE
100 CONTINUE
```

The idea behind such formatting, of course, is to make the job of pattern matching and recognition simpler for the machine. Note that the reason we do not convert to a fully normalized loop immediately, is that we do not want to lose readily-available information (for level-1 reasoning) through subscript alterations and other major changes. Now, assuming that the problem at hand is to determine whether the I2-loop can be outermosted, it is obvious, by inspection that due to the recurrence on the I2-index, the answer is *no.* In the following, we attempt to mechanize this "obvious" inference process by formulating a set of parameters and rules.

In plain english, we note that existence of I2+1 as a subscript of the left-hand-side array A and I2 as the corresponding subscript of the same variable A on the right-hand-side led us to our inference, given the additional constraint that the I2-index is an *increasing* index (not a decreasing one). Let us establish the parameter groups (L1, U1, S1) and (L2, U2, S2) to stand for the do-loop bounds on index I1 and I2 respectively. (L: lower bound, U: upper bound, S: step). These are of type *character string* and are easily read or inferred. Thus, in our case, L1 = '1', U1 = 'N', S1 = '1'; L2 = '2', U2 = 'N-1', S2 = '1'. Let int be a function which takes an argument of the above type and returns an integer. Thus int(L1) = 1; int(U1) may return 'undefined' unless the value of N is known at compile time; and so on. In case N is unknown, the user may be prompted for any attribute of N which may be needed for inferencing. For instance, the user may be asked: IS N POSITIVE? (In general, the user always has the right to respond to any such query by pressing the UNKNOWN key; in such a case, the inference mechanism should be able to proceed, exploring other avenues, if necessary, or assuming the most likely answer to the query. In dealing with such uncertain information, the final inference may have a certainty factor[17] of less than 1). A given subscript expression e1 is said to be *symbolically* greater than (less than) another subscript expression e2, if the standard polynomial form of e1 differs from that of e2 by +k (-k), where k is a known integer. Remembering that we have restricted ourselves to linear subscripts it is clear that simple pattern matching can be used to infer such symbolic relationships. Let us assume the existence of two Boolean functions sgt and slt for this purpose, where sgt(e1, e2) returns TRUE if e1 is symbolically greater than e2; slt is defined similarly.

In a more formal way than plain english, we may establish the grounds of the reasoning behind our inference by putting down a set of rules to work on a given statement, as follows:

Rule 1: IF memberof(rhs, lhs-array-variable) IS TRUE THEN
next-term = extract-from(rhs, lhs-array-variable)

Rule 2: IF memberof(second-subscript-expr(lhs), 'I2') IS TRUE
AND
memberof(second-subscript-expr(next-term), 'I2') IS TRUE
AND
sgt(second-subscript-expr(lhs),second-subscript-expr(rhs))
AND
i2-is-increasing IS TRUE THEN
recurrence-exists IS TRUE,
outermosting-possibility IS FALSE

Rule 3: IF (int(S2) > 0) OR (int(U2) > int(L2)) THEN
i2-is-increasing IS TRUE

Rule 4: IF memberof(rhs, lhs-array-variable) IS FALSE THEN quit

We have used long, self-explanatory names for parameters and functions for ease of understanding. Also, only the rules directly relevant to the particular conclusion referred to earlier, are stated. The syntax used is not exactly in accordance with that allowed by ESE. The memberof function returns TRUE if the second (string) argument matches a sub-string of the first argument. Assuming that the rules are contained within a focus control block (FCB), the action clause of Rule 4 is meant to stop further invocations of the FCB. Allowing for the possibility of multiple references of the left-hand-side array variable on the right-hand side of the assignment, recursive invocation of the FCB can be used to effectively iterate the process of rule-firings through successive updates of parameters next-term and rhs (see Rule 1).

Level-n reasoning

Continuing our discussion with respect to the loop outermosting problem, the basic question of whether a proposed outermosting is *legal,* can be solved quite effectively using a 2- or 3-level reasoning hierarchy. (The highest level must eventually perform the equivalent of a rigorous loop-interchange algorithm based on a dependence test like the Banerjee test[10,11]. however, depending on relative need, and context, it might be sufficient to do with much less). Additional clarification on this topic is available elsewhere[18].

4.0 Progress and status of EAVE

The EAVE project was conceived in April 1986. The detailed quarter-by-quarter progress since then is described in a status report[15]. The program is fully functional and prolonged testing and debugging has made it quite robust. It currently uses 110 rules, 40 FCBs and 19 external procedures. The program accepts ordinary fortran programs as input and selects the do loops sequentially for analysis and advice. Both 'interactive' and 'external file' input modes are available. Additional test/debug efforts currently envisaged are expected to be minimal.

High-level heuristics incorporated

Following is a summary of the high-level heuristics incorporated so far in EAVE.

- (1) Heuristic transformations for invocation of compound instructions (e.g., MPYADD). Certain commonly used coding disciplines used in FORTRAN inhibit the generation of such compound instructions, because of the manner of vectorization (optimization) incorporated in the VS FORTRAN, version 2 compiler. The EAVE consultant is capable of detecting such cases and suggesting suitable code transformation and rearrangement.

- (2) Transformations for use of proper do-loop indexing in nested loops to ensure efficient vector stride usage.
- (3) Transformations to enhance vector register reuse.
- (4) Various data-dependent code improvement strategies, which result in more efficient vector code generation for special cases.
- (5) Substitution of code sequences by appropriate calls to ESSL (Engineering and Scientific Subroutine Library)[2] routines.
- (6) Limited code restructuring to enhance cache reuse.

Versions of the present prototype have been made available to some internal users within IBM in order to get feedback for isolating and fixing any remaining problems.

Further work: beyond vectorization to parallelization

Currently, we are working towards development of an interactive research parallelization tool (called RPTOOL), based on the concepts in EAVE. Two distinct programming environments are being catered to: (a) the SPMD model available under EPEX FORTRAN[13,14] and (b) the Parallel FORTRAN language[19] recently announced At present, Parallel FORTRAN (PF) uses VS FORTRAN Version 2 Release .1.1 as a base, which is the programming medium catered to by the current version of EAVE. Due to the obvious scope of efficient reuse of knowledge bases already developed for EAVE, we have chosen PF as the first target in the design of RPTOOL. Our overall objective is to generalize EAVE so as to provide the best possible advice on running a given do loop in vector/parallel mode on a multi-way 3090 VF under (a) the EPEX FORTRAN SPMD model and (b) the Parallel FORTRAN language facility.

5.0 Discussion

Given *any* compiler-machine pair, there is always a potential knowledge gap between the user and the system. However, significant performance problems resulting out of this gap are beginning to surface with the introduction of the newer, high-performance vector and parallel machines only. We have sketched briefly an outline of EAVE, which is a particular solution to a very specific problem: namely, the knowledge gap between FORTRAN users and a specific compiler-machine pair. However, the *research* results obtained and leading from this exercise are basic and general enough to be able to point to future solutions to other similar problems. Due to space constraints, detailed explanations and examples of user interaction could not be provided in this short paper; the reader is referred to other reports[18].

Many of the transformations suggested and performed by EAVE, could conceivably be included in future releases of the actual compiler. However, due to the *evolutionary* change from a smart, optimizing scalar compiler to a vectorizing compiler, problems of the nature shown in this paper are bound to surface. In the absence of a *revolutionary* approach of re-engineering the compiler from scratch, it is probably quite difficult to make the compiler generate the best possible code irrespective of the programmer's coding style. Thus, the need for an EAVE-like consultant might persist for a while yet! In any case, expert tools of this flavor will probably always be useful to programmers as guides to better programming in a given environment.

References

1. IBM, VS FORTRAN Version 2 Programming Guide, February 1986; IBM order number SC26-4222.
2. IBM, Engineering and Scientific Subroutine Library Guide and Reference, February 1986; IBM order number SC23-0184.
3. IBM System/370 Vector Operations, January 1986; IBM order number SA22-7125.
4. IBM, IBM Systems Journal, Vol 25, No.1, 1986, pp. 4-81.
5. A. A. Dubrulle, R. G. Scarborough and H. G. Kolsky, "How to write good vectorizable FORTRAN," IBM Palo Alto Scientific Center, G320-3478, Sept 1985.
6. H. H. Wang, "Introduction to vectorizing techniques," IBM Palo Alto Scientific Center, G320-3489, March 1986.
7. R. G. Scarborough and H. G. Kolsky, "A vectorizing FORTRAN compiler," *IBM Journal of Research & Development*, Vol. 30, No. 2, March 1986, pp. 163-171.
8. B. Liu and N. Strother, "Peak vector performance from VS FORTRAN," IBM Technical Report # RC 12849, T. J. Watson Research Center, Yorktown Heights, NY, June 1987.
9. R. Ellersick, "Vector coding techniques for VS FORTRAN version 2," *Proceedings of SHARE 68*, March 1987.
10. U. Banerjee, S-C.Chen, D. J.Kuck and R. A. Towle, "Time and parallel processor bounds for FORTRAN-like loops," *IEEE Trans. on Computers*, Vol C-28, No. 9, September 1979, pp. 660-670.
11. J. R. Allen, "Dependence analysis for subscripted variables and its application to program transformations," Ph.D. dissertation, Rice University, 1983.
12. J. J. Dongarra, F. G. Gustavson and A. Karp, "Implementing linear algebra algorithms for dense matrices on a vector pipeline machine," *SIAM Review*, 26(1):91-112, January 1984.
13. D. A. George, "EPEX -- environment for parallel execution," in Parallel Systems and Computation, G. Paul and G. S. Almasi, ed., North-Holland, 1988.
14. F. Darema et al., "A Single-Program-Multiple-Data computational model for EPEX FORTRAN," IBM Research Report RC 11552, Yorktown Heights, NY, October 1986.
15. P. Bose, "A brief status report on EAVE: an expert adviser for vectorization," IBM Research Report RC 13353, 12/10/87.
16. P. Bose, "Gedanken experiments with FORTRAN do loops on the 3090 VF," IBM Research Report (under clearance for publication).
17. Expert System Environment/VM, Reference Manual, 1st. edition, September 1987, IBM order number SC38-7004-0.
18. P. Bose, "Interactive program improvement via EAVE: an Expert Adviser for Vectorization," *Proc. 1988 ACM Int'l. Conf. on Supercomputing*, St. Malo, France, July 1988; see also, IBM Research Report RC 13472, Yorktown Heights, NY, January 1988.
19. IBM, Parallel FORTRAN Language and Library Reference, IBM order number SC23-0431-0, March 1988.

A VISUAL PROGRAMMING ENVIRONMENT FOR THE NAVIER-STOKES COMPUTER

Sherryl Tomboulian, Thomas W. Crockett, David Middleton
Institute for Computer Applications in Science and Engineering
Mail Stop 132C NASA Langley Research Center
Hampton, VA 23665, USA

ABSTRACT

The Navier-Stokes Computer is a high-performance, reconfigurable, pipelined machine designed to solve large computational fluid dynamics problems. Due to the complexity of the architecture, development of effective high-level language compilers for the system appears to be a very difficult task. Consequently, a visual programming methodology has been developed which allows users to program the system at an architectural level by constructing diagrams of the pipeline configuration. These schematic program representations can then be checked for validity and automatically translated into machine code. The visual environment is illustrated by using a prototype graphical editor to program an example problem.

INTRODUCTION

The Navier-Stokes Computer (NSC) [6,7], developed at Princeton University with funding from NASA, is a special-purpose, high-performance parallel system designed for very large computational fluid dynamics (CFD) applications. The architecture consists of multiple processing nodes arranged in a hypercube configuration. Each node contains a few dozen functional units which can be reconfigured dynamically into one or more vector pipelines.

The architecture has some features resembling those of Multiflow's TRACE [1] computers [4] and CDC's CYBERPLUS [2] [1,3], such as multiple function units and long instruction words. However, there are significant differences which appear to make development of effective high-level language compilers a very difficult problem. As an alternative, a visual programming methodology is presented which employs a graphical interface to assist the user in programming the NSC at the machine architecture level.

A brief overview of the NSC is given first, and some of the difficulties in programming it with conventional methods are discussed. The design for a visual programming environment is then described, and a prototype version is used to illustrate the concepts for a sample problem. Conclusions based on experience with the prototype system are reported.

NSC ARCHITECTURE

The major architectural components of the Navier-Stokes Computer are described here. The focus is on the individual nodes, rather than on the system as a whole, since it is the internal design of the nodes which makes the NSC a novel architecture. The information presented is a considerable simplification of the actual design, with many details omitted for the sake of clarity. The description given is sufficient for an understanding of the visual environment described in this paper. The final design of the NSC hardware is not complete at this writing, so some adjustments to the following may be needed in the future.

Each node contains 32 functional units. Every functional unit can perform floating-point operations, and some of them can also perform either integer/logical operations or max/min computations. In addition, each functional unit has an associated register file which can be used to store constants or intermediate values, as well as to buffer data to adjust for pipeline timing delays. The functional units are hardwired into three types of ***arithmetic-logic structures*** (***ALSs***), called ***singlets***, ***doublets***, and ***triplets***, which contain respectively 1, 2, or 3 floating-point units.

Memory is arranged in 16 planes of 128 Mbytes each, for a total memory of 2 Gbytes per node. In addition, there are 16 double-buffered data caches. Two ***shift/delay units*** are provided to aid in reformatting memory data into multiple vector streams. A complex programmable switching network routes data among ALSs, memory planes, caches, and shift-delay units. Communication between nodes is handled by means of a ***hyperspace router***. The various hardware components are configured into vector pipelines during execution by programming the switches. Multiple pipelines may be set up to run in parallel. The pipeline configurations may be rapidly modified under program control as the computation proceeds through different phases. Scalars are treated as vectors of length one. A simplified diagram of the data path architecture is shown in Figure 1.

Control flow is even more complex. A central sequencer provides high-level control flow, but independent DMA controllers associated with each memory and cache plane pump data through the pipelines. An elaborate interrupt scheme is used to signal pipeline completions, evaluate conditional expressions, and trap exceptions.

Projected peak performance of the system is quite high, with a maximum rate of 640 MFLOPS per node. A 64-node NSC would have a total memory of 128 Gbytes and maximum performance of 40 GFLOPS [7].

PROGRAMMING CONSIDERATIONS

There are several features of the NSC architecture which make compilation of high-level languages into efficient object code a difficult task. One of the most serious

This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18107 while the authors were in residence at ICASE. Authors' Electronic mail: sjt@icase.arpa, tom@icase.arpa, koala@icase.arpa.

[1] *Multiflow* and *TRACE* are trademarks of Multiflow Computer, Inc.

[2] *CYBERPLUS* is a trademark of Control Data Corporation.

is the organization of memory into separate planes. During an instruction (vector operation), a function unit can read or write in only a single memory plane, and multiple function units working in the same memory plane can cause contention problems. This causes serious difficulties for a compiler in trying to decide where to allocate variables, since the optimum layout for one pipeline may be unworkable for the next. In some cases, it may be necessary to maintain multiple copies of arrays, or to relocate them between phases of the computation. Another problem arises since the function units within each ALS are not constructed identically. Only a single unit can perform integer operations, and another unit has circuitry for min/max computations. This, coupled with the distinctions between singlets, doublets, and triplets, complicates the problem of mapping function units onto expression graphs. Generation of control code is made more difficult by the presence of multiple sites of control (sequencer, DMA units, interrupts, etc.) which must be carefully orchestrated to insure that all possible actions are mutually consistent. Numerous other details tend to complicate programming as well. Any of the individual problems could probably be handled successfully, but they tend to interact with each other, making the overall problem more difficult than the sum of the subproblems. Given current compiler technology, it is difficult to see how all of these considerations can be handled simultaneously while still producing code that can achieve high utilization of 32 function units. It has been estimated that construction of a FORTRAN compiler for the NSC would take about three years, and the performance of the resulting code relative to other high-performance computers is in doubt.

Because of these problems, it seems that a programming methodology more closely tied to the architecture might deliver better performance. Traditionally, this has been accomplished by writing assembly language programs for performance-critical applications. Unfortunately, the NSC lacks anything resembling a conventional assembly language. Each instruction must be specified in a complex hierarchical microcode which contains specific control for every function unit, register file, switch setting, DMA unit, etc. The effect of an instruction is to completely specify the pipeline configuration and function unit operations for the entire machine. This requires a few thousand bits of information per instruction, encoded in dozens of separate fields. Therefore, hand-written microprograms are clearly not practical for the NSC.

A VISUAL PROGRAMMING ENVIRONMENT

In an effort to simplify programming at the architectural level, a visual programming methodology has been developed. This approach is based on an informal manual technique which evolved among applications researchers at Princeton University and NASA's Langley Research Center. Using this technique, programs were designed by hand-drawing a series of pipeline configurations, each representing one stage, or loop body, within the overall program. The natural evolution of this manual technique suggested that an automated environment in which pipeline instructions were drawn interactively on a graphics display and then automatically translated to microcode could be an effective way of programming the NSC at the machine level.

The concept of visual programming is not new, but it has become increasingly practical as workstations with high resolution graphics have become widely available [5]. A recent survey of visual programming techniques can be found in [2]. This method seems to be a natural approach for programming data flow and pipelined architectures. Visual programming techniques have been applied to parallel architectures before, but for different architectural models. A prominent example is Poker [8], which is a parallel programming environment designed to support the CHiP computer.

The scope of this project has purposely been limited to internal programming of individual nodes, since this area is the source of greatest difficulty. If needed, techniques similar to those used in Poker could be applied to the larger multi-node environment. Although the design has been tailored specifically to the NSC, the same general approach could be used for other reconfigurable pipeline machines.

Three major goals were established for the NSC visual programming environment. The first is that the representation have a one-to-one correspondence with the functional model of the machine, so that everything could be specified precisely if necessary. However, an effort would be made to choose appropriate defaults wherever possible in order to minimize the amount of detail required. The defaults could be easily overridden. The second goal is that the graphical representation be easy to program and clearly represent the semantics so that a programmer looking at an instruction would immediately see the intent. The third is that the environment would do all it could to ease the user's task by preventing or indicating syntactic errors and violations of hardware constraints as the program is entered. More extensive checking could be done when the visual representations are translated to microcode, and any additional errors would be visually presented to the programmer.

The design for the visual environment contains three major components, a graphical editor, a *checker*, and a microcode generator. Figure 2 shows the interaction between these components and the user. The graphical editor provides the usual operations found in an editor, such as the ability to enter new input, modify or delete existing data, and save the results. However in this case, the objects being operated on are graphical rather than textual. The graphical editor also is responsible for extracting information from the pictures and storing it in internal data structures. Two types of internal data are distinguished. One type consists of information which is needed solely to manage the graphical display, such as the position of images on the screen. The other type consists of semantic information which is needed in order to generate microcode. Since the semantics are represented graphically, both types of information are needed in order to reconstruct the display. But in order to generate code, only the semantic information is needed.

The checker contains, in a knowledge base or other suitable representation, detailed information about the architecture of the NSC, so far as it is relevant to the programming process. This includes various machine parameters such as the number and types of function units, their organization into ALSs, the number and size of memory planes, etc. More importantly, the checker also knows all of the rules about conflicts, constraints, asymmetries and other restrictions in the NSC architecture. The graphical editor calls on the checker at appropriate points during interaction with the user to validate the information being input. Any errors are flagged as soon as they are detected. In addition, the graphical editor uses the checker's knowledge of the architecture to reduce the possibilities for making errors. For example, if the user has routed the output from one function unit to a particular memory plane, the graphical editor will not let him send the output of a second unit to the same plane. The philosophy is similar to that embodied by syntax-directed text editors, with the goal being to assist the user in developing correct programs despite the complexity and numerous restrictions of the architecture. Another advantage of having a checker is that it helps to make the whole visual environment more robust in the face of changes to the machine design. Some changes can be handled merely by updating the knowledge base, with minimal impact on the graphical editor and microcode generator.

Once a complete program (or consistent program fragment) has been defined, the microcode generator uses the semantic data structures created by the graphical editor to generate machine code for the NSC. The checker is invoked again at this point to perform a thorough check of global constraints and other conditions which may not be practical to check during the editing process.

In order to test the concept of visual programming for the NSC, a prototype graphical editor/assembler was designed and implemented. The prototype focuses mainly on the graphical editor portion of the design, in order to determine whether the great level of detail needed in the microcode instructions can be conveniently presented pictorially, and to assess the ease of programming with this type of interface. The checker is not present as a separate entity, although some checking functions are incorporated into the graphical interface. Since the final design of the NSC is not complete, and there is no means of running actual NSC programs, the prototype produces only the semantic data structures as output, rather than the actual microcode instructions. The semantic data can be thought of as a pseudo-code representation of the instructions.

PROGRAMMING EXAMPLE

The major features of the visual environment are illustrated here using the prototype graphical editor to program an example problem. The example is a point Jacobi update for the 3-D Poisson equation on a uniform grid, with a residual convergence check. The equation for the update is given by

$$u_{ijk}^{(m+1)} = u_{ijk}^{(m)} + \frac{h^2}{6} R_{ijk}^{(m)} \qquad (1)$$

$$\begin{aligned} R_{ijk}^{(m)} &= \frac{1}{h^2}(u_{i+1,jk}^{(m)} + u_{i-1,jk}^{(m)} + u_{i,j+1,k}^{(m)} + u_{i,j-1,k}^{(m)} \qquad (2) \\ &\quad + u_{ij,k+1}^{(m)} + u_{ij,k-1}^{(m)} - 6u_{ijk}^{(m)}) - G_{ijk} \end{aligned}$$

A more complete explanation of this problem for the NSC can be found in [7].

The central concept of the system is that visual objects, or *icons*, are used to represent architectural components of the NSC at a suitable level of abstraction. The user manipulates these icons interactively to construct a program. Subimages within each icon are also meaningful, providing the interface to an additional level of program detail. A high-resolution bit-mapped display is used as the drawing surface. Interaction is provided primarily with a "mouse", augmented with a keyboard for some operations.

In the prototype, icons consist principally of the three different ALS types. Two representations of the doublet are provided, since doublets may be configured to operate as singlets by bypassing one of the functional units. Functional units are shown as squares within the icon, with the "double box" units having integer/logical as well as floating-point capabilities. Other icons which would be useful, but are not currently implemented, include memory planes and shift-delay units.

In addition to the icons, a variety of other visual devices are employed. These include pop-up menus and subwindows, "buttons", "sliders", and even text fields, where appropriate. The prototype is implemented on a Sun-3 workstation using SunView[3] graphics software.

Figure 3 shows the basic display window used. The right hand side is a "control panel" area used to select icons and specify various editor operations. The large area in the center is the drawing space in which pipeline diagrams are constructed. Informational and error messages are displayed in the narrow strip across the top. The region at the left is reserved for control flow specifications and variable declarations, which are not implemented in the prototype.

To construct a program, a user defines a series of pipeline diagrams. Each pipeline corresponds to a single instruction, or one line of code, in a more conventional language. Control panel operations provide the usual editor operations to insert, delete, copy, and renumber pipelines, as well as to scroll forward or backward or jump to a specific pipeline.

The first step in constructing a pipeline diagram is to select the needed ALSs and position them in the drawing area of the screen (Figure 3.) This is accomplished by moving the mouse pointer into the control panel area and selecting the appropriate icon, then "dragging" the outline of the ALS to its desired location. The process is repeated until all of the needed ALSs have been selected.

The next step is to specify the inputs and outputs of the function units. These are selected by "mousing" on the I/O pads (short wires terminated by small black circles). A menu pops up showing the available choices.

[3]*Sun-3* and *SunView* are trademarks of Sun Microsystems, Inc.

These may be either external connections to other function units, caches, memories, or shift/delay units, or else internal connections for feedback loops or register file data. Timing delays, needed for proper alignment of vector streams, may be introduced by routing input data into a circular queue in a register file and then retrieving the value a number of clock cycles later when it appears at the head of the queue.

Figure 4a illustrates the process of connecting the output from one function unit to the input of another. The mouse controls a "rubber-band" line which conceptually indicates a wiring connection between the two pads. The checker is used during this operation to ensure that only legal connections are attempted. The microcode generator would later derive switch settings by interrogating the connection tables built by the graphical editor.

In the case of a cache or memory connection, additional information is needed to program the DMA units. This is handled by a pop-up subwindow, in which the cache or memory plane number, variable name or starting address, stride, etc. are specified.

Note that the use of pop-up menus and windows is crucial to our approach. By hiding ancillary information until it is needed, the amount of detail displayed in the pipeline diagrams is reduced to a manageable level. Menus and subwindow templates also serve to prompt the user for needed information and remind him of his choices, both valuable services in an environment as complex as the NSC architecture.

The third and final step is to program the functional units by specifying the arithmetic or logical operations which they are to perform. Once again this is done with a pop-up menu (Figure 4b). The menu appears when the mouse is used to select a function unit within an ALS. Figure 5 shows the completed pipeline diagram for the point Jacobi iteration of Equation 1.

CONCLUSIONS

While the results based on the prototype graphical editor should be regarded as preliminary, it appears feasible to implement a complete visual programming environment for the Navier-Stokes Computer. This environment would clearly be more convenient and faster to use than hand-written microcode. The improvements derive from several factors. First, the visual representation more clearly reflects the hardware architecture and program intent than do reams of textual microassembler code. The data-flow style of the diagrams also seems to be a natural way for humans to describe computations. In addition, information hiding with subwindows can be used to effectively reduce the amount of low-level detail which must be displayed and assimilated at one time. This is somewhat analogous to the use of macros and subroutines in textual languages. Another advantage is that the detailed knowledge of architectural intricacies built into the visual environment reduces the possibility of writing erroneous programs and errors are caught sooner when they do occur.

On the other hand, programming at this level, even with the graphical interface, is a tedious process. The amount of machine-level detail which must be specified requires that the programmer have a good understanding of the hardware design. The user must focus not only on solving his problem, but also on mapping his problem onto this very complex architecture. So given a choice, a higher-level programming environment would be preferable.

One approach to reducing the complexity is to use a simpler architectural model, perhaps a subset of the NSC. The tradeoff here is between performance and programmability. By ignoring certain features of the architecture, it may become easier to program, but performance may be adversely affected in some situations. Initial examination of this approach has shown that some abstraction is possible, but the performance ramifications are unclear.

The visual environment could potentially be extended to include debugging features. During execution, each new instruction would display the corresponding pipeline diagram, annotated to show data values flowing through the pipeline. This could help to pinpoint timing errors, as well as other bugs in the program. The visual environment might also be useful as a back end to a compiler, displaying the results of the compilation process.

In summary, a visual programming environment offers several advantages for efficiently programming a reconfigurable pipeline architecture such as the NSC. However, it is still essentially a low-level programming language, and requires a significant implementation effort in order to become a useful tool. It remains to be seen whether this approach can compete with compiled high-level languages over the long term.

REFERENCES

[1] R.G. Babb, L. Storc, W.C. Ragsdale, "A Large-Grain Data Flow Scheduler for Parallel Processing on CYBERPLUS", *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 845-848.

[2] S. Chang, "Visual Languages: A Tutorial and Survey", *IEEE Software*, Vol. 4, No. 1, Jan. 1987, pp. 29-39.

[3] Control Data Corporation, "CYBERPLUS Hardware Reference Manual", Publication No. 77960981.

[4] J.A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", *Computer*, Vol. 17, No.7, July 1984, pp. 45-53.

[5] R.J.K. Jacob, "A State Transition Diagram Language for Visual Programming", *Computer*, Vol. 18, No.8, Aug. 1985, pp. 51-59.

[6] D.M. Nosenchuck, M.G. Littman, W. Flannery, "Two-Dimensional Nonsteady Viscous Flow Simulation on the Navier-Stokes Computer MiniNode", *Journal of Scientific Computing*, Vol. 1, No. 1, 1986.

[7] D.M. Nosenchuck, S.E. Krist, T.A. Zang, "On Multigrid Methods for the Navier-Stokes Computer", in *Multigrid Methods*, S. McCormick and K. Stuben (eds.), Marcel-Dekker, 1988.

[8] L. Snyder, "Parallel Programming and the Poker Programming Environment", *Computer*, Vol. 17, No.7, July 1984, pp. 27-36.

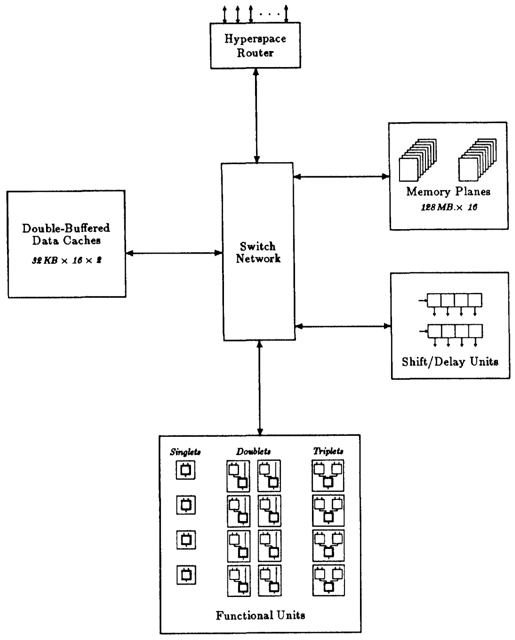


Figure 1: Simplified diagram of the datapath architecture of the Navier-Stokes Computer.

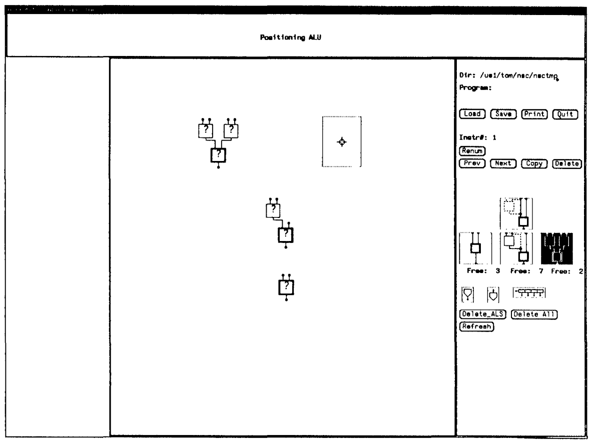
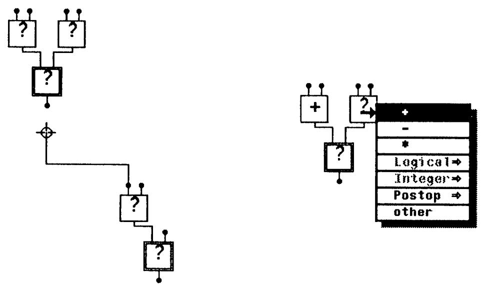


Figure 3: Selecting and positioning an icon.



(a) (b)

Figure 4: Programming function units. (a) Specifying pipeline connections. (b) Selecting operators.

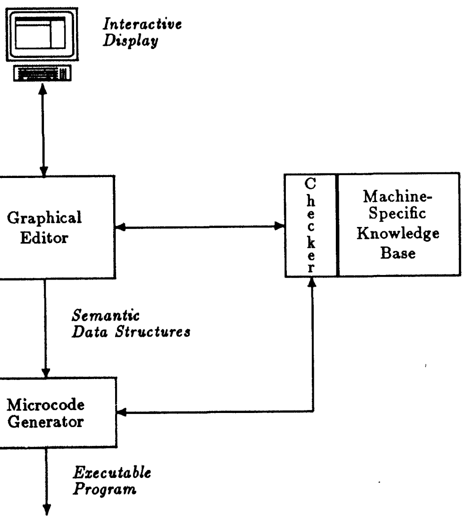


Figure 2: Major components of the visual programming system.

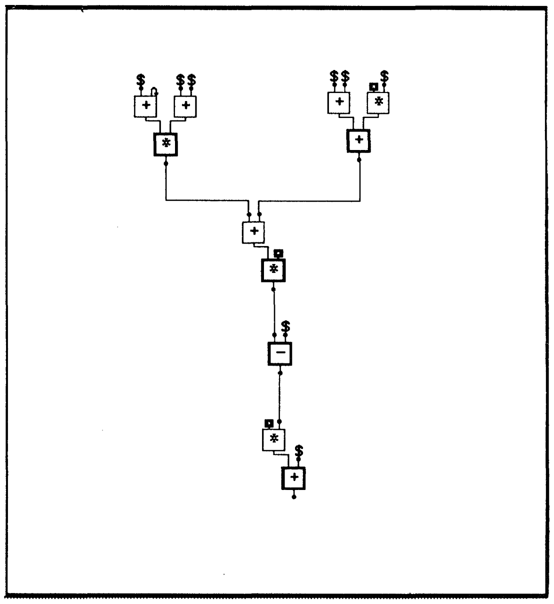


Figure 5: Completed pipeline diagram for the point Jacobi iteration.

The PFG Language: Visual Programming for Concurrent Computation

P. David Stotts

Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

Abstract

PFG (Parallel Flow Graphs) is a language for expression of concurrent, time-dependent computations. Its syntax is graphical and hierarchical to allow construction and viewing of realistically-sized programs. Its execution semantics are defined by a mathematical model of concurrent computation based on timed Petri nets and hierarchical graphs. The PFG language and underlying computation paradigm serves as the foundation of a development and analysis environment for real-time software systems under development at the University of Maryland.

PFG is rich enough to express many of the common concurrent control structures found in parallel languages, as well as some less common ones. Each syntactic structure in PFG has a direct translation into a portion of a timed Petri net model. The net created by legally combining PFG structures is guaranteed to be well-formed, in the sense that each Petri net is in the *free-choice* class and has a clear interpretation in terms of a hardware/software system. Several techniques have been defined which allow the model produced from a PFG program to be analyzed for concurrency properties, such as deadlock freedom and proper mutual exclusion on shared data structures.

1. Visual programming and concurrency

Though graphical languages are not a new idea, they have not caught on particularly well, especially in comparison to the popularity of textual languages. Part of this failure up until now may have been from the relative lack of high-resolution, bit-mapped screens for display of graphical programs. The widespread availability of desk-top workstations now eases this problem considerably. Icon-based tools, offering a pictorial style of user-interface, are now very popular and should presage a renewal of interest in graphical programming as well.

Many researchers have designed and experimented with graphical languages. Representative projects include the PICT system [3], which uses flowchart-like diagrams constructed by the user interactively, and the PROGRAPH language [5, 6] which allows interactive construction of functional dataflow programs. Earlier work on dataflow languages also used graphical program representations [2], though the user interfaces were not as visually rich as those of the more recent projects due to hardware limitations. The Poker programming environment [8] is another language that allows visual programming. It supports concurrent computations for the CHiP parallel processor architecture, but its visual interface is limited to the grid-based specification of a graph showing communication paths among the parallel processes in a program.

Though PFG is graphical in its syntax, it differs from previous visual languages in two aspects: it is intended for the expression and analysis of concurrent computations (PROGRAPH and Poker are among the few others), and its semantics are formally defined by a mathematical model based on timed Petri nets and hierarchical graphs. In essence the graphical syntax for PFG is just a convenient method for a user to specify the mathematical model of his computation. Previous graphical languages with mathematical semantic models, like FGL [4] and GPL [1], have been confined to the dataflow paradigm.

The HG model of concurrent software systems, which defines the formal semantics of PFG, is explained in section 2. This theory forms the basis for both static program analyses and dynamic analyses (in the form of execution simulation using the Petri net execution rules). Section 3 contains the definition of a parallel flow graph and and an explanation of the syntax of PFG. Section 4 then discusses the timing aspects of a PFG program. A description of the translation from PFG into the HG model is presented in section 5, and the general utility of the language is then illustrated in section 6 by giving the PFG representations of several well known concurrent control structures, and discussing several analysis techniques that have been developed for concurrent computations in PFG. Section 7 concludes with a brief discussion of future research plans using PFG.

2. The formal semantics of PFG

The formal semantic definition of the PFG language is provided by the *HG software system model.* This theory is intended for the representation and analysis of concurrent, time-dependent systems composed of a combination of software (applications, operating system, language support, etc.) and hardware (host machine). The mathematical details are presented fully elsewhere [9, 10], but for ease of discussion we present here a summary of the theory, with emphasis on the issues of concurrency.

The HG formalism separates the major aspects of concurrent computation into three distinct model components.

- The *data model* is a formal representation, using h-graphs, of the structure and interrelationships among collections of data that are to be transformed by the computation under study. The h-graphs provide enough leverage to detect overlapping access to different parts of data structures by concurrently executing code segments.
- The *static program model* is a representation of all the operations on data (procedure calls) required by a computation as a set of non-overlapping *basic blocks*. Execution of the procedure calls in each block is necessarily sequential, but blocks can execute concurrently with each other. Each procedure during execution has its own local data area, and the procedure call semantics require copy-in, copy-out argument passing. The formalism for expressing basic blocks works in conjunction with the h-graph formalism to allow complete determination of operations which alter (as opposed to simply viewing) portions of data structures.
- The model component of greatest interest here is the *control flow model*. It expresses the possible parallel execution threads of a concurrent computation. A thread is a sequence of basic blocks from the static program model, the execution of which produces the portion of the total computation contributed by that thread. The control flow model is a timed Petri net together with a (somewhat complex) interpretation of the net structure in terms of the other model components.

2.1. Data modeling

Representation of data in PFG is done with an extension to the theory of *hierarchical graphs*, or *h-graphs*, first developed by Pratt [7]. The extended theory presupposes two universal, finite base sets: the set Φ of *nodes*; and the set Ξ of *characters*. An *atom* is a finite sequence of characters from Ξ. The set of all atoms is denoted Δ, and $\Delta=\Xi^*$. The atom # denotes the null, or empty, string. An *extended directed graph* (or simply *graph*) over Φ and Δ is the standard notion of directed graph with atoms appearing as labels on the arcs. Given these, the following definition presents the concept of an h-graph, the basic model of data in this theory:

Definition 1: H-graph

An *h-graph* over Φ and Δ is a triple, $h=<G,V,r>$, in which

$G=\{g_1,\cdots,g_k\}$, $k\geq 1$, is a finite subset of Ω, such that each $g_i=<M_i,E_i>$

$V: \bigcup_{i=1}^{k} M_i \to G\cup\Delta$

$r\in G$

G is termed the *graph set* of h; V is the *immediate value function*; r is the *root graph* of h. We assume that $r=g_1$ and write $h=<G,V>$.

Related terms:

a. $\bigcup_{i=1}^{k} M_i$ is the *nodeset* of h, written $\overline{M}(h)$.

b. If $m\in\overline{M}(h)$, $V(m)$ is the *value* of m in h.

c. If $V(m)\in G$ then m is a *graph-valued node* of h; otherwise $V(m)\in\Delta$ and m is an *atom-valued node* of h.

d. The set of all h-graphs over Φ and Δ is denoted Γ.

e. The set $\Psi=\Gamma\cup\Delta$ is termed the set of *values*.

An h-graph is essentially a collection of directed graphs and atoms, and a function which maps the nodes in the graphs into these entities, thus creating a structural hierarchy among the graphs. Figure 1 illustrates these concepts.

Selection of a node from the hierarchical structure of an h-graph is performed by an *h-graph selector*, or simply *selector*. H-graph selectors are syntactically the concatenation of one or more graph path designations, with embedded indications of when the hierarchy of graphs is delved into more deeply. Semantically, a node is selected by repeating for each path designation this procedure:

- apply the path designation to the target graph, obtaining a node;
- apply the h-graph value function of the target graph to the node, obtaining a new target graph.

The selection is started by using the root graph of the h-graph as the first target graph. Considering the entire h-graph h in Figure 1, some sample h-graph selectors and their respective function values are shown in Figure 2. As in the previous selector example, the value of each selector application is the *node* designated by the outer brackets. Node values are indicated for clarity. Note that the single "/" selector denotes the top level node in the h-graph, and that the value in that node is another graph. Also, note that a selector produces a *node*; the value function must then be invoked if the value of that node is desired.

Selectors provide the link between the control flow model and the data model. At decision points in the control flow, one or more paths are selected from a set of alternatives according to the value found in nodes of the data state, as indicated by specific selectors.

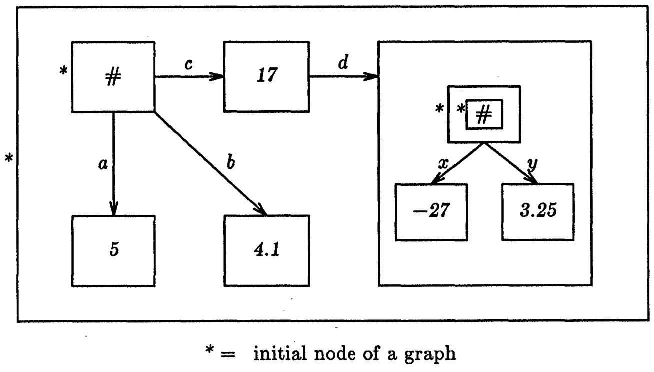


Figure 1 Example h-graph.

```
selector      node selected

/        →  [ [ # ]
               -a-> [ 5 ]
               -b-> [ 4.1 ]
               -c-> [ 17 ] -d-> [ [ [ # ] ]
                                   -x-> [ -27 ]
                                   -y-> [ 3.25 ]
                                  ]
            ]

//a      →  [ 5 ]
//c.d/x  →  [ -27 ]
//c.d//  →  [ # ]
```

Figure 2 Sample selectors for h-graph in Figure 1.

2.2. Control flow modeling

Both the structure and the semantics of Petri nets have been enhanced for modeling software. First, deterministic times have been added, one per place in the net, with each place time being an integer greater than zero. As discussed more completely later, a place represents one of the basic blocks of procedure calls in the static program model, and the time associated with a place represents the execution duration of that code block. Secondly, to model code block execution, the notion of *token aging* is included in the net execution semantics. A token arriving at a place p with time τ_p cannot participate in enabling the transitions following p until τ_p *time units* have expired, at which point the token is said to be *fully aged*. A time unit can have several definitions; the most convenient for our purpose is simply one state change of the entire Petri net. Thus the interpretation for this situation is that τ_p state changes occur in the modeled system while the software associated with p is executing.

Thirdly, to mesh token aging with the rest of the net semantics, we employ a *concurrent transition firing rule*. This allows a single state change to be effected by the firing of more than one transition. A transition is *enabled* when a fully aged token resides in each of the places that are inputs to it. A transition is *data-enabled* (in the case of two or more transitions sharing input places) if the state of the data model specifies it over the others in conflict with it. From the current state (net marking) all data-enabled transitions are identified, and some subset of them is fired. The effects of these firings are accumulated to produce the next state.

The concurrent firing rule allows state changes in the net to be equated with ticks of a wall clock. This in turn allows modeling and analysis of time-dependent computations on parallel hardware. There may be state changes in which no transitions fire, due to a lack of fully aged tokens in the net. The entire effect of such a state change is to age all tokens one "tick." The length of the Petri net state sequence produced by concurrent net execution, then, gives the duration of the modeled computation.

The data state is transformed in lock-step with the state changes in the control flow model. The control state transition rule dictates which code blocks are to be executing at any particular instant. The data state transition rule provides semantics for creating local data regions for procedures when called, passing arguments via copy-in, copy-out semantics, and effecting the function calculated by each procedure on its local data. The two rules are coordinated in that one data state change occurs for each control state change.

A word is in order here about our interpretation of these nets in terms of the hardware that is intended to host the modeled computation. Our working assumption is that each place in a Petri net is mapped to one (unique) processor in some parallel architecture; the mapping is one-to-one, but not necessarily onto. Because of the association of code blocks with places, then, each basic block executes on it own processor. While this may be unrealistic for large computations on today's machines, it may not be so for machines in the near future. It also makes analysis easier, and so is a reasonable assumption for a first look at the utility of this model. Tokens may, under this view, be thought of as requests for the hardware processor to execute its associated code block. Only one request is handled at a time, which means that only one token at a place is allowed to age. Any others arriving while this is happening simply wait their turn, in "limbo." The same code block is executed to fulfill each request. Thus, tokens have no identity, and there is no need to queue them to preserve their arrival order. Finally, a transition with a fully aged input set of tokens must fire as soon as the subset selection allows it to do so. No arbitrary waiting is allowed as in the original Petri net execution semantics. This restriction is made to ease the problems associated with timing analyses. With one processor per net place, it seems reasonable to insist that when one block execution request is satisfied, the processor not "idle", but get right to handling any other of its outstanding requests (tokens).

3. The syntax of PFG: parallel flow graphs

The control flow model, as presented, is largely a general Petri net with some additions that enhance its suitability for time-dependent analysis. The PFG language offers a technique for controlling the acceptable structure of these nets, that is, limiting the software modeler to using only a subset of the general timed Petri nets. The restrictions serve the same purpose in our theory that structured programming does for the creation of manageable algorithms--they limit achievable complexity but not expressive power.

To accomplish the goal of modeling concurrent computation with a Petri net structure of limited complexity, we view the static program model and the control flow model as a unified entity, represented in a graphical notation termed a *parallel flow graph*. A PFG program is constructed as a hierarchical collection of parallel flow graphs, and then each can be dissected into the two component models for analysis. The components of a Petri net produced from PFG are easily associated with portions of the modeled software, thus ensuring that analysis is attempted only for HG models with reasonable interpretations.

The following definition describes the mathematical structure of a parallel flow graph.

Definition 2: Parallel flow graph

Let W be a set of procedure calls, S be a set of selectors, and Υ be a distinguished node value. A *parallel flow graph* ϕ over W, S, and Υ is a tuple ϕ=<g,K,V,$\hat{t}$> in which

a. g=<η,E_ϕ,η'> where
   η is a finite set of nodes,
   $\eta' \in \eta$ is the *initial node*, and
   E_ϕ is a finite set of arcs, each $e_{\phi_i} \in E_\phi$ of the form <η_j,η_k,a> with $\eta_j, \eta_k \in \eta$ and $a \in \Delta$, indicating that an arc labeled with atom a exists from node η_j to node η_k; the arcs in E_ϕ are subject to the restrictions stated below.

b. K: $\eta \rightarrow$ { pcall, cbranch, nbranch, join } is a function mapping each node in g into one of four types, termed respectively *procedure call*, *concurrency branch*, *nondeterministic branch*, and *join*.

c. V: $\eta \rightarrow S \cup W \cup \{\Upsilon\}$ is a function mapping each node in g into either a selector, a procedure call, or the distinguished value Υ. The value Υ only serves to make the function total.

d. $\hat{t}$: $\eta \rightarrow \{1,2, \cdots\} \cup \{\infty\}$ is a function that associates a positive, integral execution time, or the value ∞, with each node in the PFG.

Visually, a parallel flow graph is drawn with different icons to represent the four node types. A *concurrency branch* is denoted by a base-down triangular icon. A *nondeterministic branch* is denoted by a base-down half-circle icon. A *join* node is denoted with a base-up triangular icon. We employ a syntactic shorthand in the case of procedure call nodes. Rather than explicitly picture each node, we represent an entire sequence of them as a single rectangular icon, termed a *basic block* node. The underlying mathematical entity still contains a sequence of individual "pcall" nodes. The node icons in a parallel flow graph are connected with arrows. The PFG prototype allows an icon to be "clicked" open to reveal its contents (value), either a selector expression or a block of procedure calls, in a viewing window. At the outer syntactic level, nodes are numbered; branch nodes are indicated by notation such as "s4", and basic blocks are indicated with notation like "b7." Join nodes are pictured with Υ on them.

We now describe restrictions on the general structure prescribed in the definition of a parallel flow graph. Because each procedure begins with a single control path, the initial node η' may not be of type "join." The arcs between nodes represent the flow of control from one action in an algorithm to the next, and each arc has an atomic label associated with it. To ensure connectivity, each node in the graph must be on a directed path from the initial node. Obviously at least one arc, then, must enter each node (other than the initial node), but we place no upper limit on this number. The initial node may possibly have no arcs entering it.

Arcs leaving a node are governed by several constraints. A node containing a procedure call or a join may have no arcs leaving it, or it may have a single arc leaving it with the label on that arc being *null*, written #[1]. Each of the two types of *branch node* contains a selector, and may have any positive number of arcs leaving it. The label on each of these arcs may be any atom from Δ, and they need not be unique, *i.e.*, an atom may serve as label for two or more of the out-arcs of a branch node. Figure 3 illustrates this synthesis with a portion of a PFG in which each s_i represents a selector and each b_i represents a basic block node.

As stated earlier, the formal semantics of PFG are fully defined by the HG computation model. Informally, this model prescribes the following computational behavior for a parallel flow graph. Execution proceeds from the initial node, and nodes are executed in the order they are encountered by following arcs. Such a sequence is termed a

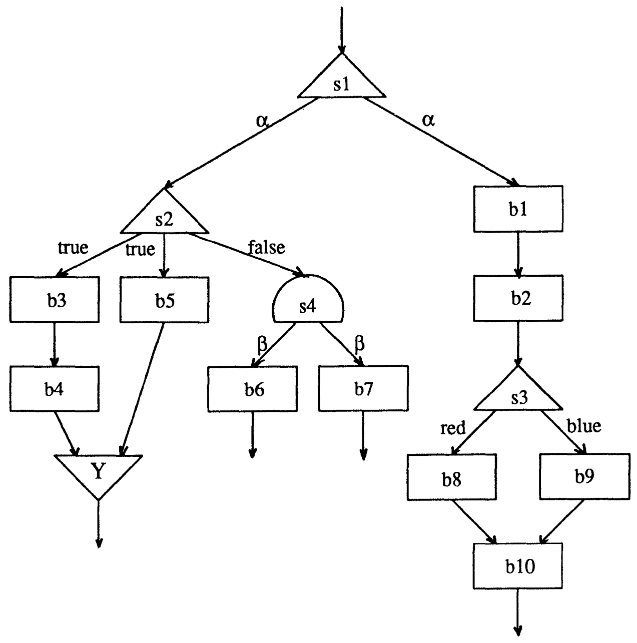


Figure 3 Parallel flow graph.

[1]By default, an arc with no written label has the *null* label.

control thread. An initial data state, represented by an h-graph, is assumed. The effects of executing a node are dependent on its type. If a node contains a procedure call, then the data state is altered as specified by the function of the called procedure. If a node contains a selector, then the data state is consulted at the node selected and a choice of next node (or nodes) is made based on the value found there. For a node of type "cbranch", two or more parallel control threads can be created. All arcs bearing the atom label found as the value of the selected node are concurrently followed. For a node of type "nbranch", the selector is evaluated to get an atom; then, *one* of the perhaps several arcs bearing that atom as label is chosen *nondeterministically* and followed. If a node contains a join 𝕐, then synchronization of the potentially many incoming concurrent control paths is performed, and a single control path continues from the node. If no arc leaves a node, or if none bears the selected atom, then the control path through that node expires; execution does not continue from the node. Execution of the entire PFG terminates when all individual control paths expire.

Since a PFG has a single initial node, the execution of a PFG always begins with one control path. When a branch node selector produces an atom that labels several out arcs, then concurrent control paths come into being. Since branch nodes are the only nodes allowed to have multiple out-arcs, they are the only points in a PFG at which concurrent control paths can be created. Subsequently, the progression of actions along each parallel control path is considered to be executing asynchronously and concurrently with the other parallel paths. Though the synchronization and merging of parallel paths is possible with 𝕐 nodes, it is not required. Two or more parallel paths may come together in a common segment of a PFG without being joined. Each path retains its separate identity and proceeds in turn to execute the PFG nodes in the common section. This feature, coupled with the fact that PFGs may be cyclic, allows a potentially unbounded number of parallel paths to be created in a computation. The number of such paths that can actually be executing at any time (as opposed to activated but waiting) is bounded, however, since the number of nodes in a PFG is finite.

Note that sequential computation is represented by a special form of parallel flow graph, one in which labels on arcs leaving a concurrency branch node must be unique. Under this restriction, at most one control thread may proceed from any node in a sequential PFG. With only one initial node, no concurrent activity can then be generated. This simple and succinct restriction adds to the attractiveness of the theory as a unified computation model.

4. Procedure timing in PFG

While timing information is an insignificant part of PFG syntax, it is an important part of the HG software system model, and hence of the semantics of PFG programs. All data transformation in PFG is accomplished via procedure calls (expression evaluation, the only other operation on the data model, is read-only and enables branching and parameter passing). A basic block in the HG model is a sequence of procedure calls unbroken by any branches. These procedures are of two kinds: *primitive*, and not. A primitive procedure in a PFG program has no parallel flow graph to represent its structure; it has only a duration (a timing) and a data transformation specified by a function. A non-primitive procedure, on the other hand, has a parallel flow graph representation of its structure. Its timing is then recursively derivable from the structure of the procedures called, with the primitive procedures providing the base timings that cause the recursion to terminate. Its function is also derivable, as the composition of the functions of the called procedures, in one of the possibly many orders specified by the control flow model.

The Petri nets employed in the HG model are determinately timed. Software is often not determinate in its behavior, that is, a block of procedure calls will often have an execution duration that varies with the input data. PFG is intended for the expression of computations in a way that will allow verification of adherence to absolute timing constraints, such as "module X must finish in under 10

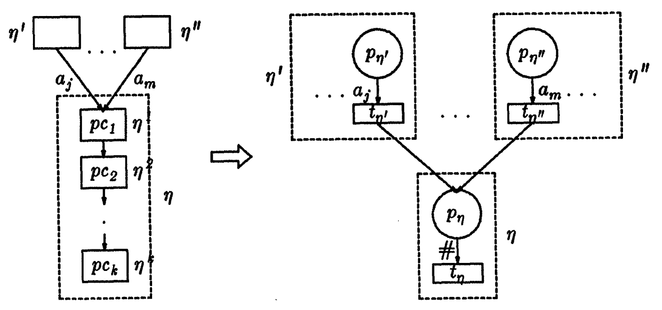


Figure 4 Translation of a basic block node.

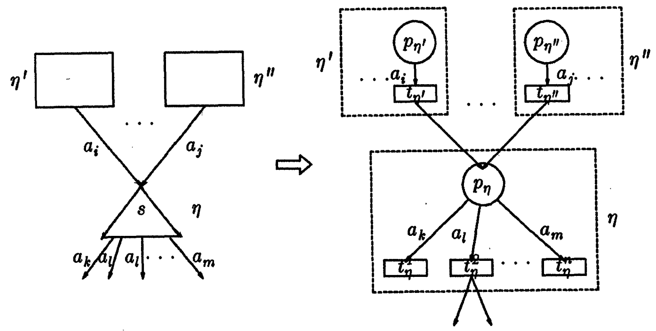


Figure 5 Translation of a cbranch node.

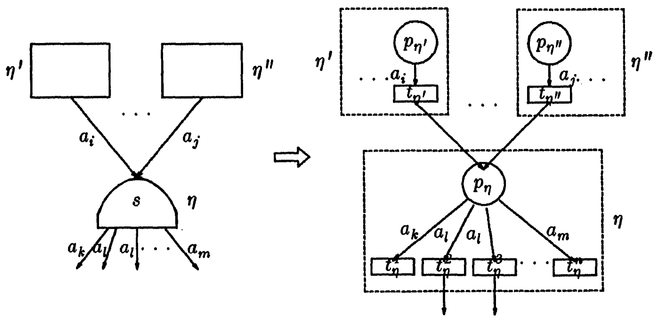


Figure 6 Translation of an nbranch node.

milliseconds," or "module Y can be no longer than 3 seconds behind module X in completion of execution."

Timing of procedures under the determinate semantics, then, is accomplished by constructing two models for each--a minimum timed model and a maximum timed model. These execution bounds are obtained by path analysis on the concurrent reachability tree [11], for the Petri net in the control flow model. This graph is a state-space structure that reflects the difference between regular Petri net semantics and the concurrent firing rule employed in PFG semantics. It is constructed in such a way that duration of a block is reflected by the length of the state sequence (path in the graph) in which it is active. With some restrictions on its structure, such as breadth-first node generation, the concurrent reachability tree can be searched for the longest and shortest paths, and the leaves of those paths can be checked for repetition of earlier states (indicating a potentially infinite duration). Naturally, for cyclic procedures, either one of these bounds can be infinite, and the duration of ∞ will propagate to procedures which call it. Thus each PFG program is analyzed as a dual-system model for timing.

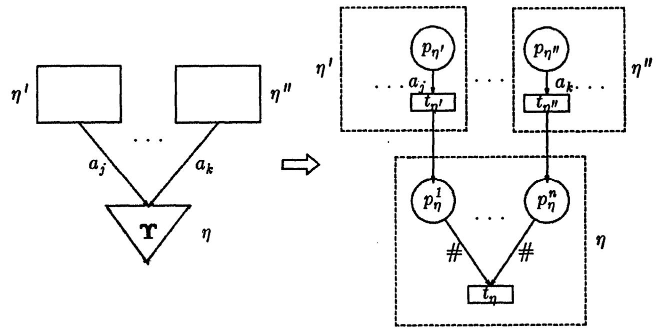


Figure 7 Translation of a join node.

5. Translation of PFG into timed Petri nets

Each structure in a parallel flow graph has a translation specified into the HG modeling formalism. Figures 4, 5, 6, and 7 illustrate the Petri net components of the control flow model, and the connections among them, created for each type of PFG structure that can be encountered in a program. The details of this translation are fully specified in [9]. In summary, an unbroken sequence of PFG procedure call nodes is coalesced into a single entity (a basic block in the static program model) and represented by a single place connected to a single transition in the Petri net (termed a *P/T* component). A branch node in a PFG program has multiple arcs leaving it, perhaps several labeled with the same atom. For a cbranch, a Petri net structure s created having a single place connected to several transitions, one for each unique atomic arc label (termed a *P/nT* component). The same number of arcs leave each transition as there are bearing its atomic label leaving the PFG branch node. For an nbranch, the translation is similar, except that one transition is created for each arc, with one arc leaving each transition. A join node in PFG has multiple arcs entering it that must be synchronized and coalesced. It becomes a Petri net structure having one place for each arc entering the join node, and a single transition to which the places are all connected (termed an *nP/T* component). A single arc then exits this transition. Once created, the Petri net components have the same interconnectivity as the PFG nodes have. The timing on a PFG node is the timing given each place in the Petri net component created from it.

The Petri nets that are created from PFG programs by this translation form a subclass of general Petri nets, termed *free-choice* nets. Their structure is simplified in that if any place serves as input to several transitions, then it is the only input place for those transitions. The transitions that share the input place are said to be in conflict. Hack [12] has

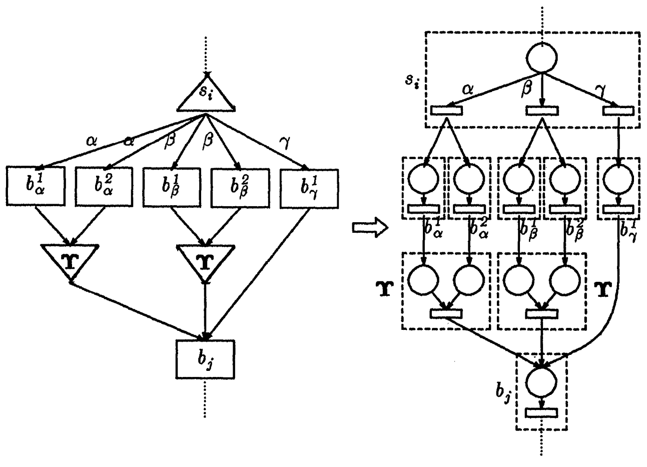


Figure 8 Multi-way fork-and-join (cobegin).

shown necessary and sufficient conditions to guarantee liveness and safeness of free-choice nets.

6. Utility of PFG

Many common concurrent control structures can easily be expressed in PFG. Figures 8 and 9 show examples along with the Petri net components created from them. Programs written to use such semantics can then be analyzed for concurrency problems using the HG model. In addition, the syntax of PFG allows expression of some concurrent control structures which have no well-known names, as exemplified in Figure 10.

Several analysis techniques have been developed for PFG programs. The dual-model method for timing of systems has been previously mentioned. It allows verification of adherence of procedures to execution time bounds. Details of this method are presented in [9].

Another analysis technique allows the detection and correction of improper accesses to shared data structures [10]. The analysis is based on the concurrent reachability tree mentioned earlier. Since a code block is associated with each place, a state (net marking) showing a token in a place indicates that a code block is executing in that state. The concurrent reachability tree, then, shows in its markings which pairs of code blocks may possibly execute concurrently. The data structures accessed by these code blocks are checked for improper accesses, such as one procedure reading twice consecutively from a datum, and another procedure concurrently writing to the same datum. The h-graphs used to model the data provide the ability to detect conflicts on portions of structured data rather than simply on variable names. When identified, these potential improper accesses can be prevented during execution by automatic insertion of synchronizing places into the Petri net models representing the calling procedure. As small an involved portion as possible of each code block is identified, and each block is restructured into two or more new, smaller blocks. These new blocks are then given Petri net places and transitions in the control flow model. For each block pair an extra place, marked with a single token, is connected into the model to create mutual exclusion on the conflicting sequences of procedure calls.

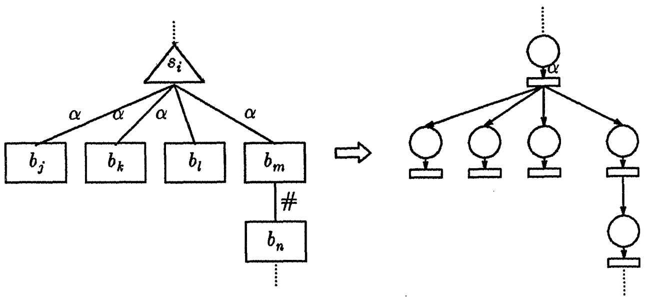


Figure 9 Spawn concurrent threads.

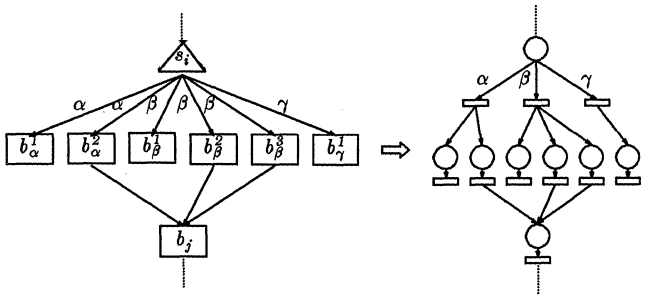


Figure 10 Arbitrary concurrent control structure.

A third analysis for the HG model is detection of some deadlocks, which appear in the concurrent reachability tree as partial markings of places that are portions of join components. The semantics of a join are such that if any one of the places entering it are marked (indicating that the software represented by that place is executing) then all of the other places must eventually be marked as well in the same state, or the following transition can never fire, blocking progress at that point. Further, the fully marked state must be reachable from the partially marked ones. For example, consider a join component having three incoming arcs in a PFG program. The join is represented in the Petri net of the control flow model by three places entering a single transition. If some state μ appears in the reachability tree having tokens in one of these places, then there must be a state μ' on a path from μ that has all three places marked. If no such μ' exists, then a potential deadlock exists in the original PFG program.

In addition to these concurrency analyses, aliasing detection in the data model has been developed by Wilson [13] for sequential computations (a special case in PFG syntax).

7. Future research

PFG is interesting both for its graphical syntax and for its formal concurrent computation semantics. The language allows expression of time-dependent concurrent computations; the underlying semantic model allows incorporation of the behavior of the host machine into analysis of the system. An initial implementation of the language is just now underway, in conjunction with the development of the PFG programming environment. The PFG environment is a unified construction and analysis toolset for concurrent, time-dependent computations. It has the HG software system model as a formal basis for all activity in the environment: static analysis, dynamic simulation, and code generation. The PFG language serves as the primary program source. Programming in other languages, such as Ada or Modula-2, is possible in the PFG environment, with source programs being translated directly into the HG model. After creation, the model can then be viewed as a PFG program. Various target machines will have HG representations stored in a modelbase so that time-dependent analyses can be done on a software system for a particular host. Once an HG model has been analyzed and is correct, executable code for the host can be generated from the model. The system is being developed for a Sun workstation.

References

1. A. L. Davis and S. A. Lowder, "A Sample Management Application Program in a Graphical Data-Driven Programming Language," *Digest of Papers, Compcon Spring 81*, pp. 162-167 (February 1981).

2. A. L. Davis and R. M. Keller, "Data Flow Program Graphs," *Computer*, **15**(2), pp. 26-41 (February 1982).

3. E. P. Glinert and S. L. Tanimoto, "Pict: An Interactive Graphical Programming Environment," *Computer*, **17**(11), pp. 7-25 (November 1984).

4. R. M. Keller and W.-C. J. Yen, "A Graphical Approach to Software Development Using Function Graphs," *Digest of Papers, Compcon Spring 81*, pp. 156-161 (February 1981).

5. T. Pietrzykowski, S. Matwin, and T. Muldner, "The Programming Language PROGRAPH: Yet Another Application of Graphics," *Graphics Interface '83*, Edmonton, Alberta, pp. 143-145 (May 1983).

6. T. Pietrzykowski and S. Matwin, "PROGRAPH: A Preliminary Report," Technical Report TR-84-07, University of Ottawa (April 1984).

7. T. W. Pratt, "Formal Specification of Software Using H-Graph Semantics," pp. 314-332, in *Lecture Notes in Computer Science #153: Graph Grammars and Their Application to Computer Science*, ed. H. Ehrig, M. Nagl, and G. Rozenberg, Springer-Verlag, Berlin (1983).

8. L. Snyder, "Parallel Programming and the Poker Programming Environment," *Computer*, **17**(7), pp. 27-36 (July 1984).

9. P. D. Stotts, Jr., "A Hierarchical Graph Model of Concurrent Real-Time Software Systems," Ph. D. Dissertation (TR-86-12), Department of Computer Science, University of Virginia, Charlottesville, Virginia (August 1985).

10. P. D. Stotts, Jr. and T. W. Pratt, "Hierarchical Modeling of Software Systems With Timed Petri Nets," *Proceedings of the International Workshop on Timed Petri Nets*, Torino, Italy, pp. 32-39 (July 1985).

11. P. D. Stotts and T. W. Pratt, "Petri Net Reachability Trees for Concurrent Execution Rules," *Journal of Parallel and Distributed Computing* (accepted, to appear).

12. M. Hack, "Analysis of Production Schemata by Petri Nets," *M.S. thesis*, Cambridge, Massachusetts, Department of Electrical Engineering, Massachusetts Institute of Technology (February 1972).

13. J. N. Wilson, "Data Types and Aliasing in Program Specification and Verification," Ph. D. dissertation (TR-86-13), University of Virginia, Department of Computer Science, Charlottesville, Virginia (May 1985).

ARCHITECTURE AND LANGUAGE INDEPENDENT PARALLEL PROGRAMMING: A FEASIBILITY DEMONSTRATION *

S. Sobek, M. Azam, and J.C. Browne
Department of Computer Sciences
The University of Texas
Austin, TX 78712-1188

Abstract

This paper describes a system for development of architecturally independent parallel programs. The concept bases for the programming system include the separation of specification of dependency relations from specification of units of computation and the formalization of specifications for dependency relations so that they can be readily translated to a spectrum of implementation mechanisms. The host for the implementation is a Sun workstation. The languages in which units of computation can be expressed include Ada, C and Fortran. The targets for execution of the parallel programs include a Sequent Balance, a VAX cluster, an Intel Hypercube, and a Cray XMP. The graphical/visual user interface to the programming system has been found to be a major contributor to its effectiveness.

1 INTRODUCTION

This paper presents an environment for parallel programming in which are prepared programs which can execute on a variety of multiprocessor architectures without user modification. A parallel program is viewed as a set of units of computation composed into a computation by dependency relations which specify the order of execution of the units of computation. The user-visible structure of a program developed in this environment is independent of the programming languages in which the component modules of the program are written. The component modules may be a mixture of several different languages, including Ada, C and Fortran. The key conceptual principles are separation of specification of units of computation from specification of dependency relations and formal specification of dependency relations at a level of abstraction which allows ready translation to a variety of synchronization and/or communication mechanisms. Program structure is expressed in a declarative hierarchy which allows effective application of architecturally specific optimizing compilers to the component modules. The user interface is graphical and at a level of abstraction which promotes effective program formulation as well as preparation of executable programs.

The programming system, Computation-Oriented Display Environment or CODE, has been in operation for some months and has been used to generate parallel programs for several different programming environments. Complete architectural independence is not a claim. Although the unified model of parallel computation underpinning this work covers SIMD models, the implemented system covers only MIMD architectures and is practically limited to large grain parallelism.

This approach to parallel programming is complementary to the automatic restructuring of existing higher level language programs. Statement and loop level parallelization can still be done on the module level components of the parallel structure created with CODE.

2 APPROACH

The approach we have taken is to define a unified model of parallel computation at a level of abstraction which can be easily mapped to a spectrum of architectural mechanisms for implementation of dependency relations. Browne [BRO85, BRO86] has given an informal description of this unified model, while Sobek [SOB88] has given a full and formal definition of the unified model.

There are two particularly significant properties of this representation of parallel computations. First, it is sufficiently formal and complete to support translation to representation at greater levels of resolution. That is, a parseable formal grammar can be written for the dependency relations and the firing rules specified in the unified model. Second, the representation of dependency relations is cleanly separated from the representation of computations. The result of these two properties is that a program becomes a computation structure specified as a set of computation modules and the dependency relationships among this set of modules. The only intrinsic limitation placed on the target execution system is that the data, control and constraint (shared name access control) dependencies of the unified model of dependency relations must be expressible in the mechanisms provided.

Separating specification of units of computation from specification of dependency relations and the provision for several levels of abstraction in specifying units of computation enable **the definition of parallel computations as declarative hierarchies**. A unit of computation at one level of abstraction may itself have an arbitrarily complex sequential or parallel structure without this structure impacting the relationships between this module and the balance of the computation struc-

*This research was supported in part by the DOE under grant DE-FG05-85-ER-25010, in part by the DARPA under grant N00039-86-C-0167, and in part by the ONR under grant N00014-86-K-0763.

ture at the higher level of abstraction. The computation modules (units of computation) may thus be arbitrary programs in any high level programming language supported in an execution environment or may themselves be complex parallel computation structures. Thus, optimizing compilers which do automatic restructuring to obtain parallelism on a particular architecture may still be applied with profit to this architecturally independent parallel computation structure.

3 IMPLEMENTATION

3.1 Graphical Interface

The Computation-Oriented Display Environment (CODE) is used to develop graph structured representations of parallel computations, from which are produced the declarative specifications mentioned preceding. These specifications are parsed to produce programs in high level languages that support parallel execution. After a program is written (by drawing and annotating a graph on a graphics screen), its CODE intermediate language file and the user's code files are transferred to the desired parallel architecture. Target code is produced on the parallel machine from the transferred files.

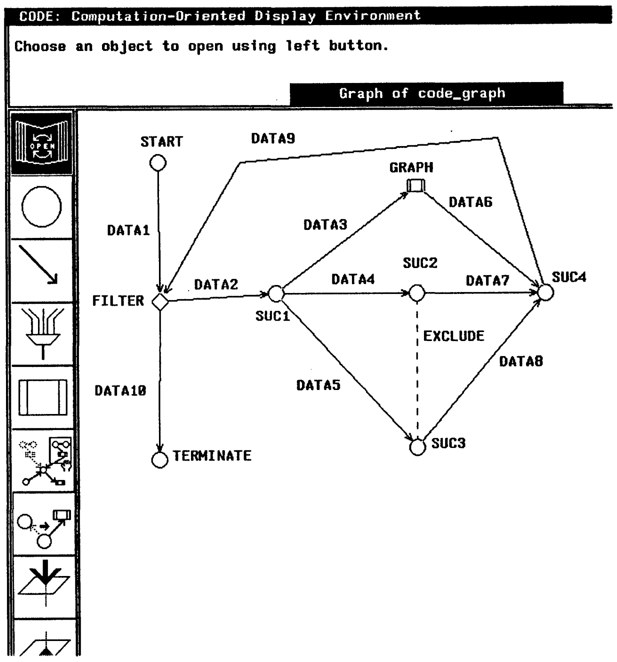
The initial version of CODE has been implemented on a Sun Microsystems workstation. CODE is written in C and uses the SunView (trademark of Sun Microsystems) package for displays.

An example CODE display is shown in Figure 1. The horizontal strip on the top is the message window. This window displays instructions, information, warning messages, and error messages, and it allows some textual input from the programmer. The vertical strip on the left hand side contains a menu window with symbols that depict major CODE commands. The remaining large drawing window on the right is used for displaying and drawing the graphical program.

A CODE program graph is composed of four main object types: schedulable units of computation (SUCs), dependencies, filters, and subgraphs. A subscripted name such as *S[5]* denotes an array of objects.

A *SUC* is associated with a subprogram written in some high level language, such as an Ada procedure. SUCs communicate via *dependencies*. The data structure associated with a dependency may be a simple variable or an array. The two kinds of dependencies implemented are data dependencies and exclusion dependencies. A data dependency totally orders the execution of the precisely two SUCs it connects. An exclusion dependency provides a partial ordering (or no ordering at all when it is used only to share data) among two or more SUCs.

A *filter* is a special computation unit with semantics defined by an associated constraint expression that defines transmission of some subset of the data from its input dependencies to some subset of its output dependen-


CODE: Computation-Oriented Display Environment

Choose an object to open using left button.

Graph of code_graph

Figure 1: Example CODE Display

cies. The constraint may be satisfied by mere existence of input data or by the actual values associated with input data dependency variables. Among the important uses of filters are for loop control and, as we describe later, acting as data dependent firing rules for SUCs.

A *subgraph* is composed of SUCs, filters, dependencies, and subgraphs. A CODE computation graph is a hierarchy in which each subgraph has a single parent graph, and the overall graph has no parent. Thus, any computation graph with a hierarchy of graphs can be transformed into a one-level graph. Dependencies may span subgraphs. Thus, a dependency in the graph level currently displayed that appears to end at a subgraph may, in fact, end at a SUC or filter arbitrarily deep in the graph hierarchy.

Briefly, a typical program development scenario using CODE would be to draw the computation as a graph of SUCs, dependencies, filters and subgraphs, set dependency properties (e.g., number of data elements) and filter constraints, type the code associated with each SUC, have CODE generate the declarations file needed to produce an executable parallel program, ship the declarations file and SUC code files to a target parallel architecture, and generate the target executable parallel program.

The set of objects and their properties were chosen so that the information necessary to synchronize and schedule operations is highly visible. In particular, we have identified commonly used constraints on execution and made specification of those constraints an integral part of CODE.

The primarily used SUC execution constraint is that all input data be available. Like many before us, we use a directed arrow (data dependency) from graph node *A* to node *B* to mean that *B* requires data produced by *A*, and thus execution of *B* may not start until execution of *A* has ended. An exclusion dependency denotes that two or more SUCs share data. Here the synchronization may not be simple, and so we have defined an exclusion constraint syntax, and implemented a subset of the syntactically correct constraints. Exclusion constraints specify under what circumstances a SUC that is otherwise able to begin execution is precluded by the current execution of some set of SUCs. For example, the constraint '*(A OR B) MUTEX C*' means that if either SUC *A* or SUC *B* is executing, then SUC *C* may not start execution, and vice versa. *A* and *B* might be only reading shared data and may execute together, while *C* might be updating that same data, and thus must execute alone.

The constraints of filters can make data-dependent SUC execution decisions quite visible. Filters can use the existence or actual values of data associated with incoming data dependencies to schedule SUC execution. Filters have their own constraint syntax. For example, assume that data dependency *D1* ends at a filter whose output data is passed to SUC *A*. The constraint '*D1[5] > 8.0*' means that for SUC *A* to begin execution, the fifth element of *D1* must be greater than 8.0.

When objects are created, they are assigned default properties. For example, the default SUC code file language might be Ada. *Options windows* are displays which show the current or default properties of objects in the program graph and allow the user to change selected properties.

The SUC options window contains the important command 'update code', which generates header and trailer lines for the SUC code file. Although a complete separation of the SUC synchronization information from the SUC module code is desirable, certain practical considerations preclude it. SUCs communicate via dependencies, and the set of dependencies associated with an existing SUC may change at any time during development. Dependencies have associated data types, and the data types of existing dependencies may change at any time during development. Requiring the user to keep track of all these changes so that later an executable program on a target system may be created is a prescription for disaster. Therefore, the 'update code' command is used to generate each SUC's connection to the rest of the parallel program (such as an Ada procedure header), and CODE checks to make sure that the user has not changed the generated header and trailer lines.

3.2 Encapsulation Strategy

Once the user is satisfied with the program graph and associated SUC code files, she or he requests that CODE generate a declarative specification of the graph. The declaration file and the SUC code files are then shipped to one or more target parallel architectures.

The end product desired is a computation in which generated code, which satisfies all declared synchronization constraints, controls the execution of user-written SUCs. On each target machine, TOAD (Translator Of A Declaration) uses synchronization information in the declarations file and target-architecture-specific information to produce a compilable program that encapsulates the SUCs.

The encapsulation must be able to pass input data to SUCs, get result data from SUCs, and execute SUCs. Although even assembly language could be encapsulated, it is easier if each SUC code language provides support, such as procedures that can access and change parameters or common data. The encapsulation must ensure that each SUC which is coded in an acceptable language is provided with the interface to other SUCs it requires.

Although the declarative specification language is architecturally independent, and **generated** headers and trailers are as portable as the language in which the SUC is written, the programming environment does not attempt to ensure the portability of **user-written** code in SUCs. Moreover, it is the user's responsibility to ensure that all information required by the encapsulation in the form of output values from SUCs is actually produced by the SUCs.

We have a TOAD for several parallal systems, including a Sequent Balance, a VAX cluster, an Intel Hypercube, and a Cray XMP. The following discusses the TOAD implemented on a Sequent Balance 21000. For that implementation, we have chosen to allow SUCs to be coded in Ada, Fortran, and C. The encapsulation code generated is an Ada program that calls SUCs as Ada subprograms, or uses the Ada pragma 'INTERFACE' for SUCs that are not Ada code.

Each non-loop data dependency has a task which accepts notification each time a SUC produces a value for that dependency. The task then notifies the proper SUC. For example, if SUC *A[3]* produces values for dependency *D[6]* to pass to SUC *B[2]*, then the task must know which element of *A* produces which 2 elements of *DEP* and which 3 elements of *DEP* are consumed by each element of *B*. Later, buffering will be added to these tasks.

Each exclusion dependency has an Ada task. The only constraints implemented enforce a multiple readers, single writer paradigm, and any exclusion is mutual exclusion. So constraints such as '*A OR B*' (denoting only readers) and '*(A OR B) MUTEX C MUTEX D*' (A and B are readers, while C and D are writers) are allowed, while '*(A AND B) MUTEX C MUTEX D*' is not be allowed. The exclusion dependency task has a reader's semaphore and a writer's semaphore. It is called by the SUC task of every associated SUC that is otherwise ready to begin execution. Later, buffering will be added to these tasks.

Each SUC has its own task that ensures that all data needed is available, all filter constraints are met, and exclusion constraints are satisfiable. It calls the user-written SUC code, and, if there are any associated dependencies, it notifies the dependencies that it has completed execution.

All tasks are terminated when all tasks are idle, and all messages queues are empty.

3.3 Execution Constraint Specification

Achieving a useful separation of the code inside SUCs from constraints on the execution of those SUCs requires careful specification of the set of allowed constraints on execution. Dependencies and filters are used by CODE and TOADs to provide very general mechanisms for specifying constraints. Dependencies can affect the execution order of two or more SUCs, while the filters of a SUC allow data dependent constraints on that SUC's execution.

An exclusion dependency is used to share data or provide a partial ordering among its two or more associated SUCs. The basic syntax of an exclusion dependency constraint is:

(*<or_suc_phrase>*) |
<mutex_phrase> MUTEX <suc_phrase> |
<suc_phrase> EX <suc_phrase>

For example, the constraint '(SA OR SB OR SC)' means any combination of the SUCs *SA*, *SB*, and *SC* may execute concurrently, and '(SA OR SB) EX SC' means if *SA* or *SB* is executing, then *SC* may not start execution.

A SUC may execute when all of its un-*filter*ed dependencies have associated data and the constraints of all of its filters are satisfied. A filter constraint may be satisfied by dependencies simply having data or by the values to which dependencies' associated data have been set. A filter constraint specifies (explicitly or implicitly): what subset of dependencies must provide data to satisfy the constraint, the constraint the above dependencies must satisfy, what subset of dependencies will be used to provide input data to the SUC when it executes, and what subset of dependencies will be consumed by filter constraint satisfaction evaluation or by SUC execution.

The basic syntax of a filter constraint is:
<dependency_expression> | *<selection_phrase>* |
[<selection_phrase>] : <dependency_expression>
[<number_of_times>]

<dependency_expression> is an expression whose truth value may be based on the values of dependencies' data. *<selection_phrase>* allows selection of a subset of the dependencies to provide filter output data. The keywords *ANY*, *HIGHEST*, *LOWEST*, *FIRST*, and *LAST* are currently allowed in the phrase, but only *ANY* is fully implemented in TOADs. *<number_of_times>* gives a criterion for when a particular dependency may that provided input data may again provide input data.

For example, consider a filter with input dependencies *DA*, *DB*, and *DC*, and the data structure associated with each is an array 1..100 of real numbers. The constraint 'ANY 2 : $$[5] > 1.0' means that once at least two dependencies have data whose fifth element is greater than 1.0, allow execution with those two as input data ($$ signifies any incoming dependency).

4 CONCLUSIONS AND FURTHER WORK

CODE and TOAD have been demonstrated to synthesize correct code for the unfamiliar and difficult portion of parallel programs, dependency relations and firing rules, for any environment for which TOAD has been implemented. Implementation effort for an initial version of TOAD in any given execution environment will be a matter of man-weeks or a few man-months at most. CODE provides a parallel structuring concept environment which is sufficiently rich to facilitate formulation of most parallel algorithms. CODE and TOAD have demonstrated the feasibility of architectural and component language independence for parallel programs.

There remain, however, a host of research issues and implementation issues. Enrichment and/or simplification of the model of computation is an open question. Will this methodology extend to SIMD architectures or is it even needed in this domain? Synthesis of efficient code for dependency relations and firing rules is a major issue. What are the issues in extending CODE and TOAD to smaller units of granularity for computational components? CODE and TOAD will be integrated with a capability for generating dependency graphs for existing programs and with a library facility for computation modules which will support both entry of existing modules and selection of modules from the library. These two features will make CODE and TOAD much more usable by providing access to modules from existing programs.

References

[BRO85] J. Browne, "Formulation and Programming of Parallel Computations: A Unified Approach", *Proc. of 1985 Int. Conf. on Parallel Processing*, University Park, Pennsylvania, August 1985, 624-631.

[BRO86] J. Browne, "Framework for Formulation and Analysis of Parallel Computation Structures", *Parallel Computing 3*, 1986, 1-9.

[SOB88] S. Sobek, "Architecture Independent Parallel Programming", Ph. D. dissertation, Department of Computer Sciences, The University of Texas, Austin, Texas, to be published in August 1988.

Communication and code optimization in SIMD programs

Allan L. Fisher Peter T. Highnam
Carnegie Mellon University Department of Computer Science
Pittsburgh, Pennsylvania 15213

Abstract

The use of SIMD architectures has been impeded by the lack of efficient high-level programming languages. This paper describes an abstraction of communication in SIMD systems that is similar in spirit to previous proposals, but whose functional formulation allows the application of powerful code optimization techniques. We give examples of how a fairly simple compiler can produce SIMD code of handcrafted quality. We further show how programs expressed in this style can conveniently be mapped to alternate topologies and execution schedules.

1. Introduction

The SIMD model of computation that we address consists of an array of identical processing elements with a regular interconnection network. The processing elements receive a broadcast instruction from a global controller, which can also perform scalar computations independently of the array. Initially, we will consider a 2-dimensional SIMD mesh with connections between each cell and its orthogonal nearest neighbors. Later, we give an example of how our system of communication primitives can be used to aid in the mapping of such a virtual grid onto a machine with a different topology.

The programming style we advocate assumes that a parallel decomposition of the task has already been performed. In the area of image processing, from which we draw the code examples used in this paper, this usually involves performing the same computation at each pixel in an image. This approach has been described in earlier work[1], with access to neighboring values specified using relative coordinates. Our approach, instead, is to use a functional notation for communication, which in turn allows the detection of redundant computation among neighboring cells.

This research is supported by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-87-C-1499.

2. Directionals

A directional is a unary operator that permits position-independent code to reference data held at other locations specified in a relative manner. Our grid model uses four directionals: **LEFT**, **RIGHT**, **UP** and **DOWN**. The semantics, using **LEFT** as a generic example, are:

LEFT *expr*	The value of this expression is the result of the computation that represents *expr* as computed in the left neighbor.

Directionals can be applied to the result of any expression, in particular they may be applied to expressions that involve directionals. For example, we can access the upper-left diagonal neighbor's p value using **LEFT UP** p, or equivalently **UP LEFT** p. (We note that similar operators can be defined for other topologies including trees, hypercubes, shuffle graphs, etc.)

More formally, we can define an algebra of operations on grids of values, with the usual arithmetic and relational operators applied point-by-point, and directionals representing translations. Where a finite grid is used, we leave border semantics up to the implementation.

The important properties of directionals from the viewpoint of code optimization are their interactions with each other and with other operators. An optimizing compiler can make extensive use of algebraic properties; the properties of directionals are listed below. Syntactically, we have defined directionals to be prefix operators of precedence equal to that of unary minus. Semantically, prefixing an expression with a directional in a SIMD architecture is a way of saying that the computation makes use of work done at another site.

The pairs **LEFT**, **RIGHT** and **UP**, **DOWN** have equivalent and orthogonal semantics. In the following, the **LEFT-RIGHT** pair is used generically; α represents a possibly empty sequence of directionals. Along with the description of mathematical properties, we describe computational motivation for their use.

- **LEFT** *expr*. The value of *expr* is computed at the left neighbor and the result is transferred here.
- **LEFT** α = α **LEFT**. The ordering of a sequence of directionals is unimportant. Permuting a sequence of directionals simply changes the routing that a value takes to achieve a fixed relative translation. (This is true on a grid, but not true of all directional algebras.)
- *Cancellation.* When a **LEFT** and a **RIGHT** directional are present in a directional sequence α, we can delete both because their net effect is a zero transfer.
- *Directionals and unary operators.* Applying a non-directional unary operator *unop* to the result of an expression *expr* prefixed by a directional **LEFT** means that *expr* is evaluated on the left neighbor, the result transferred here and then the *unop* is applied to the result. For reasons that will become clear later we would prefer the neighbor to do all of the computation and transmit the result to us. The resulting value, *unop expr* and its location are the same:

  unop **LEFT** *expr* = **LEFT** *unop expr*
- *Directionals and binary operators.* When both of a binary operator's operands are prefixed by the same directional we are specifying that the two operand values be evaluated at another PE; the values moved here from that PE and then the binary operation is performed. We would prefer the neighbor who evaluated the operands to also carry out the operation and transfer that result to us, at a savings of one communication operation. The resulting value and its location are the same:

  (**LEFT** $expr_1$) *binop* (**LEFT** $expr_2$)
  = **LEFT** ($expr_1$ *binop* $expr_2$)
- **LEFT** *constant*. Applying a directional to a constant that has the same value in all PEs is a wasted operation and can simply be removed:

  LEFT *constant* = *constant*
- **LEFT** *globalvariable*. The value of a global variable is either not used in the computation of any PE, or it is broadcast to all PEs. In the former instance applying a directional to the value is incorrect. Applying a directional to a value that is broadcast is equivalent to supplying a constant to all PEs and the directional is redundant:

  LEFT *globalvariable* = *globalvariable*

A programmer would like to be able to freely mix constants, local and global variables together in a program. Part of the compiler's task can therefore be expected to be determining the site of particular components of the code. Because the controller in a SIMD machine is typically able to execute code concurrently with the PEs, accurately extracting the maximum amount of controller computation is an important optimization. With these considerations in mind we introduce a fifth directional, not available to the programmer, but used within the compiler: **NOMOVE**. A constant or global variable within the forest used by the compiler to represent the input program is internally prefixed with this new directional. The semantics are simple: the value of the expression argument to a **NOMOVE** is computed only from constants and global variables, and hence may be computed in the controller at run-time (or possibly by constant-folding during compilation). The addition of the new directional has the further desirable attribute of reducing the special cases to be handled during compilation. The directional property list given above is augmented as follows:

- **LEFT NOMOVE** *expr*. When a directional is applied to the result of an expression given by a **NOMOVE** we know that the directional is redundant and can be removed:

  LEFT NOMOVE *expr* = **NOMOVE** *expr*
- **NOMOVE** behaves with unary and binary operators as a regular directional.
- **LEFT**, **NOMOVE** and binary operators. When the two operands of a binary operator are values produced by a **NOMOVE** (*operand*$_1$) and a regular directional (*operand*$_2$) then the computation described is: compute *operand*$_2$ at another PE; move the value of *operand*$_2$ as described by the directional; perform the binary operation on the global value. We can treat the binary operator with this mix of operands as the equivalent unary operator with one operand (*operand*$_1$) fixed as a constant:

  (**NOMOVE** $expr_1$) *binop* (**LEFT** $expr_2$)
  = **LEFT** ((**NOMOVE** $expr_1$) *binop* $expr_2$)

3. Experiments

Two translators have been implemented as a preliminary assessment of directionals as a language feature and of the leverage gained by conventional compiler optimization techniques[2] using the information provided by directionals.

The first translator implements an extension of the C language[3], *SASS*, for writing position independent code using directionals. The translator consists of a preprocessor to the standard C compiler, including interfacing to the standard CMU image routine libraries. Both PE local and global variables are supported. Using C as the target language permits straightforward mechanisms such as matrices to be used to represent processing state at each position in the image grid, and directionals are reduced to integer relative offsets for use in array index computations. As demonstrated by the *SASS* pseudo-median filtering program in Figure 3-1, grid algorithms are naturally expressed.

The second translator is an optimizer for expression graphs including directionals, based on algebraic transformation and common subexpression elimination. Such an optimizer is useful as a machine-independent stage in a complete SIMD compiler. The optimizer receives a data-dependency graph of a single-assignment language program without looping. It performs optimizations on the forest, producing a modified graph that could be used for code generation. The program was written in the OPS5 language[4], which allows straightforward expression and modification of graph-theoretic transformations. The remainder of this section describes the results achieved by the optimizer on some code fragments; the structure of the optimizer itself is described in Section 4.

```
; Three variables instantiated at every pixel position.
local(pixel)
local(colmedian)
local(result)


prog(
  colmedian = median( pixel,
                      up(pixel),
                      down(pixel) ) ;
  result    = median( colmedian,
                      left(colmedian),
                      right(colmedian) )
   )
```

Figure 3-1: *SASS* example: 3x3 pseudo-median.

We present four code fragments, first as written without regard to efficiency, and then as improved by the optimizer. For readability, we have omitted all **NOMOVE** directionals. In each case, the optimized code is as good as that produced by an experienced SIMD programmer (ignoring machine-dependencies). While a complete assessment of this technique will require examination of a wide range of programs, these results are representative of our testing of a dozen or so code fragments.

1. Finding the maximum value in the linear five neighborhood of a point nominally involves six lateral communication steps. The transformation demonstrated in Figure I-1 saves two communication steps.

2. Summing the 2×2 region of which the point is the lower righthand corner is entered as three additions and four communication operations (Figure I-2). The compiler transformations reduced this to two additions and two communication steps.

3. A linear filter applied as a convolution can be treated as the other examples to achieve some savings. In this example however the optimizer is supplied with code for a 3×3 convolution filter that is two-way symmetric. As a result of the filter's symmetry, the computation uses only three global variables for filter coefficients. The operation specified has nine multiplications, eight additions and twelve communication steps (Figure I-3). The optimizations produce an equivalent expression with three multiplications, eight additions (as before) and six communication steps.

4. As a final example, Figure I-4 shows edge-finding code transliterated directly from a robot navigation system in use at CMU, along with an intermediate version following algebraic transformations and a final optimized version. The code shown uses a 3×3 window, where windows up to 49×49 are often used; the optimizer produces even better results on larger windows. For this example, additions and subtractions are reduced from 17 to 9; multiplications from 26 to 8; and moves from 24 to 8. The single square root operation remains unchanged. Although the original code already makes use of symmetry properties to improve performance over a completely naive implementation, the directional formulation reveals the possibility for significant further improvement.

Our experiments, together with the experience of *Apply*[1] programmers, indicate that the point-independent coding style is appropriate for many image processing applications. The use of directionals permits concise coding, and conventional compiler optimization techniques are able to exploit the information provided by directionals to produce high-quality SIMD code. Furthermore, the compilation complexity (as measured by the number of major decisions taken in the course of the compilation) for each of the examples we have listed was extremely low.

Most of the expressive power of directionals is also provided by the relative coordinates used in Apply and earlier notations. Although we find directionals syntactically sweeter in many cases, their main advantage is in simplifying optimization. Without explicit use of a directional-like algebra, equivalent optimization of expressions using relative coordinates would require the use of an algebra of general arithmetic expressions, a much harder task.

4. A SIMD code optimizer

The objective of the optimizer is to minimize the number of operations that will be performed by the system at runtime, where an operation is a PE instruction or an inter-cell communication step. No other considerations are taken into account during the compilation; in particular, storage management is ignored. Section 5 contains some discussion of other issues.

The optimizer implements a straightforward greedy algorithm that performs algebraic transformations till exhaustion, followed by a common subexpression elimination stage. Not only does the optimizer attempt to minimize runtime but (via the **NOMOVE** mechanism) it also flags expressions that may be computed at compile time or entirely within the global controller. The forest received by the optimizer is essentially a basic block. The optimizer (by the nature of its implementation medium) is easily extended to make use of additional information on the operators and structures it must deal with. For example, point-independent code that reports a single value as a function of its neighborhood in many cases makes use of fully associative operators. The optimizer can be informed that an operator is associative by a one line statement supplied with the program.

Common subexpression elimination (CSE), the penultimate stage of a compiler (and the last in which machine-independent optimizations usually occur), changes the representation of the program graph, not the logical structure. This permits us to divide the optimizer structure as shown without loss of potential optimizations. In SIMD programs, locating common subexpressions below different directionals is particularly useful, as in Figure I-3.

5. Machine dependencies

In this section, we examine the mapping of a virtual grid program using directionals onto a vector of processors[5, 6, 7]. (Such a program, of course, can be implemented directly or via multiplexing on a machine of matching topology[5, 8, 9, 10, 11].)

One mapping of a virtual grid onto a vector is based on sweeping the vector over the image. Therefore, unlike the grid, algorithms written for a vector cannot assume that the entire input is available for immediate reference when the computation begins. Suppose the vector is considered to sweep down the image (i.e., the image is delivered to the vector one row at a time, beginning with the topmost row). The compiler can apply this schedule by introducing directionals representing time: replace **UP** with **PREV**, and **DOWN** with **NEXT**. The **NEXT** directional cannot be implemented directly, of course, except by prescient hardware. This problem is solved by performing the following operation on the program graph, before the optimization phase:

1. Select those nodes that represent assignments to variables, the roots of distinct expression trees.
2. Find the largest number of **NEXT** directionals that occur between one of the nodes selected in the previous stage and a leaf, *k*.
3. Insert *k* **PREV** directionals immediately above each leaf.

The graph produced by the optimization will not contain any **NEXT** directionals. **PREV** expressions can be implemented using circular buffers local to each processing element.

An alternative mapping can be applied in cases where data dependencies extend across an entire image row, as is typical of many "scan-line" algorithms. If the image is skewed so that pixels in column *i* arrive one time unit before their row-mates in column *i*+1, scan-line processing is effectively pipelined over many lines, and full processor utilization can be achieved. Program graphs with this feature can be recognized automatically by examining dependency loops, and the mapping can be implemented by the following replacements (before optimization begins):

- **LEFT** becomes **PREV LEFT**.
- **RIGHT** becomes **NEXT RIGHT**.
- **UP** becomes **PREV**.
- **DOWN** becomes **NEXT**.

6. Conclusions

We have presented an abstraction of communication in fine-grain SIMD machines, and have shown how to apply it to the optimization and scheduling of parallel programs. A simple algebraic substitution pass, followed by common subexpression elimination, is highly effective in detecting and eliminating redundant computation and communication among groups of neighboring cells. The use of directional operators also simplifies the task of mapping "virtual" topologies onto real machines.

We have not addressed some other issues of practical importance. One such issue is optimization between basic blocks; loop unrolling and functional embedding of conditionals will broaden the scope of what can be optimized. Another issue is the introduction of more precise measures of operation cost; here, we expect a fairly simple formulation to give good results. A complete assessment of our approach awaits the construction of a complete translator from a high-level language to machine code. The construction of such a compiler for the CMU Scan Line Array Processor[6] is underway.

References

1. L. G. C. Hamey, J. A. Webb and I.-C. Wu, "Low-level vision on Warp and the Apply programming model", in *Parallel Computation and Computers for Artificial Intelligence,* J. Kowalik, ed., Kluwer Academic Publishers, 1987.
2. A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, 1977.
3. B. W. Kernighan and D. M. Ritchie, *The C programming language,* Prentice-Hall, 1978.
4. C. L. Forgy, "OPS5 User's Manual", Technical Report CMU-CS-81-135, Computer Science Department, Carnegie Mellon University, July 1981.
5. M. J. B. Duff and T. J. Fountain, *Cellular Logic Image Processing,* Academic Press, 1986.
6. A. L. Fisher and P. T. Highnam, "Real-Time Image Processing on Scan Line Array Processors", *IEEE Computer Society Workshop on Computer Architectures for Pattern Analysis and Image Database Management,* November 1985.

7. S. S. Wilson, "The PIXIE-5000 - A systolic array processor", *IEEE Computer Society Workshop on Computer Architectures for Pattern Analysis and Image Database Management,* November 1985, pp. 477-483.

8. K. E. Batcher, "Design of a Massively Parallel Processor", *IEEE Transactions on Computers,* Vol. C-29, No. 9, September 1981, pp. 836-840.

9. P. M. Flanders, D. J. Hunt, S. F. Reddaway and D. Parkinson, "Efficient high speed computing with the Distributed Array Processor", in *High speed computer and algorithm organization,* D. J. Kuck, D. H. Lawrie and A. H. Sameh, eds., Academic Press, 1977, pp. 113-127.

10. W. D. Hillis, *The Connection Machine,* MIT Press, Cambridge, Massachussetts, 1985.

11. R. M. Hord, *The Illiac IV, the First Supercomputer,* Computer Science Press, 1982.

I. Optimization examples

In order to save space the four directionals **LEFT**, **RIGHT**, **DOWN** and **UP** are shown in the examples as **L**, **R**, **D** and **U**, respectively. The interested reader may find it helpful to construct the data dependency graphs that space limitations oblige us to omit.

Input:

```
result := MAX( L  L p ,
          MAX( L p ,
          MAX( p ,
          MAX( R p ,
               R R p ))))
```

Result:

```
result := MAX( p ,
          MAX( R MAX( p , R p ) ,
               L MAX( p , L  p )))
```

Figure I-1: Optimization example: (1) Max of linear five.

Input:

```
result := ( p + L p ) +
          ( U p + L U p )
```

Result:

```
temp   := p + U p;
result := temp + L temp;
```

Figure I-2: Optimization example: (2) 2x2 summation.

Input:

```
global  a,b,c;
result :=
  a * L U p +  b * U p + a * R U p +
  b * L p   +  c * p   + b * R p   +
  a * L D p +  b * D p + a * R D p;
```

Result:

```
temp1  := p * a;
temp2  := p * b;
temp3  := temp2 + U temp1 + D temp1;
result:=   R temp3 + L temp3
         + D temp2 + U temp2
         + p * c;
```

Figure I-3: Optimization example: (3) 3x3 symm. filter.

Input:

```
gx :=
  g1 * ( gpr1 * U R p + gpr0 * U p - gpr1 * U L p )
+ g0 * ( gpr1 * R p   + gpr0 * p   - gpr1 * L p )
+ g1 * ( gpr1 * D R p + gpr0 * D p - gpr1 * D L p )

gy :=
  g1 * ( gpr1 * D L p + gpr0 * L p - gpr1 * U L p )
+ g0 * ( gpr1 * D p   + gpr0 * p   - gpr1 * U p )
+ g1 * ( gpr1 * D R p + gpr0 * R p - gpr1 * U R p )

mag := sqrt(gx * gx + gy * gy)
```

Intermediate:

```
gx :=
  g1 * U ( gpr1 * R p + gpr0 * p - gpr1 * L p )
+ g0 *   ( gpr1 * R p + gpr0 * p - gpr1 * L p )
+ g1 * D ( gpr1 * R p + gpr0 * p - gpr1 * L p )

gy :=
  g1 * L ( gpr1 * D p + gpr0 * p - gpr1 * U p )
+ g0 *   ( gpr1 * D p + gpr0 * p - gpr1 * U p )
+ g1 * R ( gpr1 * D p + gpr0 * p - gpr1 * U p )

mag := sqrt( gx * gx + gy * gy )
```

Result:

```
t1  := gpr1 * p
t2  := gpr0 * p
t3  := t2 - L t1
t4  := t3 + R t1
t5  := g1 * t4
t6  := g0 * t4
t7  := t2 - U t1
t8  := t7 + D t1
t9  := g1 * t8
t10 := g0 * t8

gx  := D t5 + t6 + U t5
gy  := L t9 + t10 + R t9

mag := sqrt(gx * gx + gy * gy)
```

Figure I-4: Optimization example: (4) 3x3 edge finder.

THE DESIGN AND DEVELOPMENT OF A BASIS, α_L, FOR FORMAL FUNCTIONAL PROGRAMMING LANGUAGES WITH ARRAYS BASED ON A Mathematics of Arrays©

Lenore M. Restifo Mullin
Ashok Krishnamurthi and Deepa Iyengar

CASE Center
The New York State Center for Advanced Technology
in Computer Applications and Software Engineering
and
Northeast Parallel Architectures Center
Syracuse University
Syracuse, New York 13244

Abstract

Backus describes a formal functional programming system as a functional programming language with an associated algebra. We present in this report the design and development of the first (outer product) in a class (partitioning, structuring and orienting) of operations on arrays based on an n-dimensional indexing function ψ. These operations are the basis, α_L, for formal functional programming languages with arrays. These operations are described using *A Mathematics of Arrays* which is the algebra[(a)] of programs for a formal functional programming language with arrays. *A Mathematics of Arrays* axiomatically describes all of its arrays operations in terms of their dimensions. When we describe an arrays operations in terms of their structure we can exploit parallelism naturally, thus α_L describes and verifies the execution of concurrent operations for parallel architectures. This report describes designs and implementations for Parallel Architectures in general. In particular we address the formal design and implementation of outer-product on an Alliant FX\8 and Encore Multimax. *A Mathematics of Arrays* is based on Iverson's array operations. Iverson's array operations are based[(b)] on Sylvester's *Constructive Theory of Partitions* and Cayley's *Theory of Linear Transformations*. Sylvester and Cayley's collaborative work complemented the classical approach to differential and projective geometry developed by Riemann et al during the 19th century. We use *A Mathematics of Arrays* to govern the design and implementation of all array operations as well as partitioning algorithms at both the coarse and fine grained level. We are thus using our theory to govern the design and development of a scheduler as well as an operating system for parallel languages on parallel architectures.

Preface

The purpose of this report is to document the development of α_L, a basis for formal functional programming languages with arrays, based on *A Mathematics of Arrays. A Mathematics of Arrays* axiomatizes a functional indexing enhanced subset of Iverson's[9] structuring and partitioning operations. Iverson's work[(b)] is based on J. Sylvester's *Constructive Theory of Partitions*[19] and A. Cayley's *Theory of Linear Transformatioins*[4]. Cayley and Sylvester's development of algebraic geometry complemented the classical work in differential and projective geometry at a time when Riemann lived. Cayley first gave the definition of an abstract group. Sylvester first introduced the idea of *A Universal Algebra* based on array operations which was referenced and furthered by Whitehead at the end of the 19th century. It is *A Universal Algebra* that will become the *Algebra of Programs* for α_L, the basis for formal functional programs with arrays. *A Mathematics of Arrays* is the dissertation topic of the first author (School of Computer Science - Syracuse University).

We view the axioms and theorems in *A Mathematics of Arrays* relating to structure as an axiomatic basis for an n-dimensional geometry with symmetries. The motion of these structures over time is based on array expressions containing structuring, partitioning, and orienting operations. *A Mathematics of Arrays* axiomatically describes all partitioning operations and linear transformations on arrays in terms of their shape and the n-dimensional indexing function ψ. An axiomatic definition provides a basis for symbolic reasoning about arrays. Thus, we can use the axioms and theorems in *A Mathematics of Arrays* as rewrite rules in mechanical theorem provers based on resolution with unification and equality[6] to verify array expressions. *A Mathematics of Arrays* describes operations and combining forms on arrays. Combining forms can allow high level programs to build still higher level ones in a style not possible in conventional languages[3]. An axiomatic definition of arrays provides the basis for reasoning about arrays in general and may be used to reason about arrays in any formal functional programming language. Thus, *A Mathematics of Arrays* is an algebra of programs for formal functional languages with arrays. We call this basis α_L. All array operations as well as processor and vector register allocation(coarse and fine grained partitioning) will be executed in parallel and are direct implementations of the theory. Symbolic computation and numeric computation go hand and hand, thus we can reason both symbolically and numerically about array problems.

Introduction

We showed in CASE Report No. 8712[13] that although the structuring operations are a subset of Iverson's APL language[9], their design and implementation are different than the same operations in *A Mathematics of Arrays.* If the same APL operations were parallelized they would still not meet the theoretical designs used in *A Mathematics of Arrays* because each structuring operation in *A Mathematics of Arrays* is described using an array shape in conjunction with ψ. We will show in future reports how the structuring, partitioning, and orienting operations are variations of one higher level operation.

We have eliminated all but the structuring, partitioning and orienting operations from the C written APL\11. That means, only expression evaluation is possible from the skeleton from which α_L is being developed. We assume all function definition, assignment and all functional language environments are handled by λ definitions. Eventually, α_L can be included in any programming language environment. All of our designs as well as their implementations will be based on *A Mathematics of Arrays.* We will show how we are developing a scheduler and thus an operating system based on *A Mathematics of Arrays.* Non-functional languages, including all dialects of APL, as well as procedural languages such as Ada, Fortran, PL1, Pascal, etc., can include our compiled operations and thus benefit from the speed-ups.

Some of, our goals include, describing all of the structuring operations in terms of one high level operation and developing simplification of array expressions as theorems in *A Mathematics of Arrays* which could be used in a compiler for α_L. We plan to develop an environment which can handle array operations on any

© Dissertation Research - L.M.R. Mullin - School of CIS Syracuse University

(a) Sylvester's description of a Universal Algebra based on arrays was later referenced by Whitehead in *A Treatise on Universal Algebra.*

(b) conversations with K.E. Iverson - Toronto, Canada - 1987 parallel architecture, both coarse and fine grained. Our present research is on an Alliant FX\8 and Encore Multimax.

We have chosen C as the implementation language in a Unix environment because of its sound theoretical basis[(c)].

Notation: ξ - denotes an arbitary array - when a superscript is omitted ξ refers to arrays of all ranks.

ξ^n - denotes an arbitrary n-dimensional array - the rank of ξ^n is the one element vector containing the number of dimensions in ξ^n i.e. $\rho\rho\xi^n$ is $<n>$.

$\rho\xi^n$ - denotes the shape of an array. Shape is always a vector. Each component of a shape vector is the length of each of ξ^n's n dimensions. Thus $\rho<n>$, the shape of the rank vector, is <1>, the one element vector containing 1.

$,\xi^n$ - linearizes, vectorizes, ravels an array in row major order.

Outer Product - A Structuring Operation

We want to prove that array operations based on *A Mathematics of Arrays* are faster and more space efficient than any array operations or concurrency operations provided by Encore or Alliant and that our designs are optimal and general for both architectures. Our experiments focus on the execution of an outerproduct with two array operands. In all of our experiments we monitor the performance of an outerproduct with 400 component operands and a 160000 component result(for integer and floating point operations).

Let L_{op_+} denote outer product as an infix operator with the scalar operation plus. Therefore,

$$\xi_l \; L_{op_+} \; \xi_r$$

is the operation we want to perform. Ideally we want to partition this operation over ρ,ξ_l processors to obtain maximum medium-grained parallelism and do ρ,ξ_r scalar operations with each component of $,\xi_l$ in vector registers(or array processors) that are ρ,ξ_r long. We want to perform the operation such that each partial result is put into shared memory based on which unit of work was performed.

Let ι denote a number generating function such that ιn generates the numbers 0 to n-1,

$$\text{For all } i \;\epsilon\; \iota\rho,\xi_l$$
$$\xi_l \; L_{op_+} \; \xi_r$$
$$\text{is equivalent to}$$
$$(<i> \psi ,\xi_l) + ,\xi_r$$
$$\text{where the resultant shape is}$$
$$(\rho\xi_l),(\rho\xi_r)$$

The Architectures Targeted

We targeted the Alliant FX/8 and Encore Multimax for experiments. They supported UMAX and CONCENTRIX operating systems respectively.

We focus on medium-grained parallelism because our parallel processing takes place within a single application program. The Multimax and Alliant both provide software and hardware support for medium grained parallelism.We use system calls, *fork, pfork, pjoin,* and concurrency instructions to create multiple threads of control.

Arrays: The Goal

As previously mentioned, we ideally want to distribute ρ,ξ_l units of work which we will denote by $<W>$, to ρ,ξ_l processors. Each unit of work, $i \epsilon \iota W$, executes a portion of the outer product based on i. $\rho\xi_r$ segments of the result are computed asynchronously in parallel. In all of our experiments, $W = 400$. We will differentiate units of work from processes so as not to imply that we are dealing with W UNIX processes.We used *A Mathematics of Arrays* to govern the design of this scheduling algorithm. We will subsequently show how our implementation of this scheduler is described by $<W> \rho \iota R$ where R represents the number of real processors.

(c) Discussed by J. C. Browne in a talk entitled *A Unified Approach to Parallel Programming* 2 December 1987 at Syracuse University.

First, we'll describe our initial attempts at implementing $<W>\rho\iota R$. We'll then show how that implementation did not meet it's specification, $<W>\rho\,\iota R$. Finally, we will show through an informal proof, that our implementation is equivalent to the design and conjecture about how a class of loops is equivalent to $<W>\rho\,\iota R$.

We found that we received the best speed ups when we scheduled the 400 units of work over 18 real processors as opposed to letting the operating system schedule it. We refer to the system imposed local limit for processes as the number of virtual processors we could FORK to. Our method of partitioning can be viewed as building a large parallel aysynchronous software pipeline. We break the work up over the 18 real processors and sequentially run groups of work over them.

Our first set of experiments, illustrated in Figures 1a and 1b, proves that this method is the best way to partition work, W, over R real processors. That is to say,

$$\textit{Scheduling of Work} = \; <W> \rho \; \iota R$$

We will now elaborate on this definition and show how it is implemented in C. We will also show how our implementation is based on *A Mathematics of Arrays* by giving an informal proof.

Partitioning and Scheduling of Array Operations

For all $i \;\epsilon\; \iota\rho,\xi_l$, each unit of work, $(<i> \psi ,\xi_l) + ,\xi_r$, updates a portion of the result of the outer product, $<(i \times \rho,\xi_r) + j> \psi ,\xi_{result}$, for all $j \;\epsilon\; \iota\rho,\xi_r$. We denote each of these units of work as w_i, $i \epsilon \iota W$.

$$\text{for all } \; i \epsilon \; \iota\rho,\xi_l \text{ and } j \;\epsilon\; \iota\rho,\xi_r$$
$$w_i \text{ is}$$
$$<(i \times\rho,\xi_r) + j>\psi \;\; ,\xi_{result} \longleftrightarrow (<i> \psi ,\xi_l) \; + \; <j> \psi,\xi_r$$

As noted previously, we partition the work, W, over R real processors in the following way:

$$\text{Scheduling of Work} = <W>\rho \; \iota R$$

A small example illustrating our designs follows:

Example:

Suppose $\xi_l = <4\ 5> \rho\ \iota 20$, ξ_r is $<3\ 4> \rho\ \iota 12$, and $R = 8$. Therefore ρ,ξ_l is <20> and W is 20. Based on our definition for outer product,

$$\xi_l \, L_{op_+} \, \xi_r$$
$$\longleftrightarrow$$

$$\overset{w_0}{((<0>\psi<0\ 1\ \cdots\ 19>) + <0\ 1\ \cdots\ 11>)},\ldots,\overset{w_{19}}{(<19>\psi<0\ 1\ \cdots\ 19>) + <0\ 1\ \cdots\ 11>}$$

with a resultant shape of

$$(<4\ 5>,<3\ 4>).$$

Now, to schedule the 20 units of work over 8 processors we want to pipeline the work as follows:

Table 1 - Processors

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19				

The above table, illustrates that we want processor 0 to handle work units 0, 8, and 16; processor 1 to handle units 1, 9, and 17; ... ;

and processor 7 units 7 and 15.

We want to do this because $\langle W\rangle\rho\,\iota R$ is $\langle 20\rangle\rho\,\iota 8\rangle$ which is $\langle 0\,1\,2\,3\,4\,5\,6\,7\,0\,1\,2\,3\,4\,5\,6\,7\,0\,1\,2\,3\rangle$. We order the processors, R, from 0 to $R-1$, based on the firing order of forks, and associate each processor number with it's locations in the vector above which has 20 components. 0's are in locations 0, 8, and 16, ... , and 7's are in locations 7 and 15. We can now relate this to the processor table above. That is, Processor 0 runs work units 0, 8, and 16, ... , and Processor 7 runs work units 7 and 15. This may be more clearly understood by viewing the following:

Let the p_j's refer to which processor we want to use and the w_i's, the unit of work we want associated with each p_j.

for $j \epsilon\ \iota 8$ and $i \epsilon\ \iota 20$

$p_j=$	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
$w_i=$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

This means that we are assigning work units 0 through 19 to processors 0 through 7 which get cyclically used until all 20 units of work are assigned.

During implementation, we wanted to assign work to Processors in groups. We did not want to assign w_0 to p_0 followed by w_1 to p_1, etc. This implied *fork*ing of work units w_0 to w_7 to processors p_0 to p_7 then returning to w_8 to repeat the cycle until all work is done. *Pfork* on the Encore Multimax had characteristics similar to this which is why it proved to be so slow compared to our pipelined approach. The extra overhead caused by *W joins* was excessive. We realized that to be efficient, we needed to build a large pipeline for the work to be processed amongst the processors. Thus we had to determine how to send groups of work to processors in a way that preserved our design.

We first tried to implement $\langle W\rangle\rho\,\iota R$ directly via the formal design:

$$\text{for all } i \epsilon\ \iota W$$
$$\langle W\rangle\rho\,\iota R$$
$$\longleftrightarrow$$
$$(\langle i\rangle \mathit{modulo}\ R)\psi\,\iota R$$
with a resultant shape of $\langle W\rangle$

This design dictated W divides which we wanted to avoid. We therefore, decided to do an implementation which first performed, $W\ \mathit{modulo}\ R$, to determine if W was evenly divisible by R. If it was evenly divisible, we equally divided the work over the processors. If it was not evenly divisible, we performed the leftover work on a single processor. The remaining work was divided evenly amongst the R processors.

Continuing with our example, we will show how this technique assigned work to processors. Since $20\ \mathit{modulo}\ 8$ is 4, we assign w_0 through w_3 to a processor. With this, we know that 16 units of work, w_4 through w_{19} can be evenly divided amongst the 8 processors. We therefore assigned w_4 and w_5 to p_0,..., and w_{18} and w_{19} to p_7. Although we eliminated the W divides, which the theory dictated, we introduced a dependency on one more processor to run the remainder of work after we determined if the work was evenly divisible by the processors.

We then realized that what we wanted to do was assign work to processors in the following way:

$p_j=$	0	0	0	1	1	1	2	2	2	3	3	3	4	4	5	5	6	6	7	7
$w_i=$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Given the firing order 0 to 19, we had to assign an ordering to the w_i's so that the correct segment of the result is updated based on the theory. If we order the w_i in the following way, we can show that our design is theoretically correct. Using the p_j's and w_i's above and an ordering which relates the w_i's to Table 1 above, denoted by ξ_{o_i}, we have the following:

$p_j=$	0	0	0	1	1	1	2	2	2	3	3	3	4	4	5	5	6	6	7	7
$w_i=$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$\xi_{o_i}=$	0	8	16	1	9	17	2	10	18	3	11	19	4	12	5	13	6	14	7	15

So if we associate the w_i's with its adjacent ξ_{o_i}, the ordering above, we can see that the correct location in the result will get updated, thus conforming to our theory. We define a unary function O such that $O(\xi^1)$ obtains the locations of ξ^1's ordered components. Therefore, using our example, if we order the components in $\langle W\rangle\rho\,\iota R$,

$$O(\langle 20\rangle\rho\,\iota 8) \longleftrightarrow \langle 0\ 8\ 16\ 1\ 9\ 17\ 2\ 10\ 18\ 3\ 11\ 19\ 4\ 12\ 5\ 13\ 6\ 14\ 7\ 15\rangle$$

ordering this we get

$$O(O(\langle 20\rangle\rho\,\iota 8)) \longleftrightarrow \langle 0\ 3\ 6\ 9\ 12\ 14\ 16\ 18\ 1\ 4\ 7\ 10\ 13\ 15\ 17\ 19\ 2\ 5\ 8\ 11\rangle$$

If we now use the result of this last ordering to index the vector associating processors to work, the p_i's, which we will denote by ξ^1_C, we see that:

$$\xi^1_C \longleftrightarrow \langle 0\ 0\ 0\ 1\ 1\ 1\ 2\ 2\ 2\ 3\ 3\ 3\ 4\ 4\ 5\ 5\ 6\ 6\ 7\ 7\rangle$$

Note that the result of $\langle 20\rangle\rho\,\iota 8$ indexed by $O(\langle 20\rangle\rho\,\iota 8)$ is $\langle 20\rangle\rho\,\iota 8$ sorted and that is ξ^1_C.

Now, we index ξ^1_C by $O(O(\langle 20\rangle\rho\,\iota 8))$ and get

$$\langle 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 0\ 1\ 2\ 3\rangle$$

which is
$\langle 20\rangle\rho\,\iota 8$

That is, if we order $\langle 20\rangle\rho\,\iota 8$ where

$\langle 20\rangle\rho\,\iota 8 \longleftrightarrow \langle 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 0\ 1\ 2\ 3\rangle$

we get

$O(\langle 20\rangle\rho\,\iota 8) \longleftrightarrow \langle 0\ 8\ 16\ 1\ 9\ 17\ 2\ 10\ 18\ 3\ 11\ 19\ 4\ 12\ 5\ 13\ 6\ 14\ 7\ 15\rangle$

which when used to index $\langle 20\rangle\rho\,\iota 8$ is

$$\langle 0\ 0\ 0\ 1\ 1\ 1\ 2\ 2\ 2\ 3\ 3\ 3\ 4\ 4\ 5\ 5\ 6\ 6\ 7\ 7\rangle$$

which is ξ^1_C above, the sorted vector we want, and is the association we want to make. We can therefore say formally that $\langle W\rangle\rho\,\iota R$ indexed by its ordering is equivalent to the program shell illustrated in *Conjecture 0* because:

Given O, we denote $\langle W\rangle\rho\,\iota R$ by $\xi^1_{\langle W\rangle\rho\,\iota R}$:

Proposition 0

$$\xi^1_{\langle W\rangle\rho\,\iota R} \longleftrightarrow (O(O(\xi^1_{\langle W\rangle\rho\,\iota R})))\ \psi\ O(\xi^1_{\langle W\rangle\rho\,\iota R})\ \psi\ \xi^1_{\langle W\rangle\rho\,\iota R}$$

Noting that ψ is an associative operation

We now conjecture given that the above Proposition is true:

Conjecture 0

For all non-negative integers w and r
Loops with the following characteristics are equivalent to
$\langle w\rangle\rho\,\iota r$

```
while ( k = 0 & k < r){
    while ( i = k & i < w){
        do i = i + r;
    }
  do ( k = k+ 1);
}
```

Note also that Proposition 0 describes the permutations that ιW goes through to return to ιW, all the valid indices of $W\rho\,\iota R$. An initial lemma in our theory states that $\iota\rho\xi^1$ indexing ξ^1 is an identity for ξ^1.

The outer loop in Conjecture 0 describes $\langle w\rangle\rho\,\iota r$ sorted, while the inner loop describes how to index $<w>\rho\ \iota r$ sorted in a way that would result in $<w>\rho\ \iota r$.

Experiments Conducted

We want to prove that partitioning of arrays and scheduling of work to processors using *A Mathematics of Arrays* is faster than any concurrency operations supported by the Encore Multimax and the Alliant FX/8.

We conducted the following experiments:

(1) - equals with equals - *fork*ing with scalar operations (integer and floating point) and shared memory on the Alliant and Encore(see Figure 2,3,4).

(a) 4, 5 MIP CE's on the Alliant compared to

(b) 10, 2 MIP CE's on the Encore

(2) - best with best - (integer and floating point)

(c) - complex of 4, 5 MIP attached/detached CE's running Fortran concurrent outer and vector inner(see Figure 2,3) compared with

(d) Encore's 18, 2 MIP processors running C and *pfork* (see Figure 4) compared with

(e) partitioning based on *A Mathematics of Arrays*

(e.1) - *fork*ing to 18 processors on Encore using scalar operations (see Figure 4) and

(e.2) - *fork*ing to 4 attached/detached CE's on the Alliant with vector operations using Alliant's Vector Registers(vector inner). (see Figure 2,3)

(3) - various configurations on the Alliant compared with what Alliant believes to be the best; Fortran running on a complex of 4 CEs using concurrent outer and vector inner support. Control of partitioning for g and h below is based on *A Mathematics of Arrays* while k and l are controlled by Concurrency Instructions in Concentrix C (integer and floating point).

(g) - *fork* to 4 attached/detached CE's - scalar inner (FOSI).(see Figure 2,3).

(h) - *fork* to 4 attached/detached CE's - vector inner (FOVI).(see Figure 2,3).

(k) - *Concurrent* outer and scalar inner running on 4 attached/detached CEs (COSI). (see Figure 2,3).

(l) - *Concurrent* outer and vector inner running on 4 attached/detached CEs. (COVI). (see Figure 2,3).

Experimental Findings

Performance Analysis

We found in both sets of initial experiments, that the Encore Multimax performed better when we did not use all of its real processors. The best performance was observed when we forked just once. As we increased the number of forks from 1 to 18 we observed a decrease in performance by a factor of ≈ 30. From 18 to 50 forks, a further degradation occured but not to such an extent as from 1 to 18 forks on 18 dedicated processors. The ideal speed-up should have been 18 forks performed on 18 real dedicated processors.

In previous experiments[13] we noticed that the sequential code running in a multi-processor environment achieved a speed-up of ≈ n where n was the number of real processors as compared to the same code running on a uniprocessor. The sequential code took ≈ 12 seconds vs 120μ seconds when we forked once and 3657μ seconds when we forked n=18 times using designs based on *A Mathematics of Arrays*. The experiments detailed in this report show that redesigning the sequential outerproduct algorithm in APL\II[10] in terms of partitioning operations based on *A Mathematics of Arrays* achieves speed-ups in excess of 100 times as compared to running the sequential code on 18 real processors.

UMAX appears to incur a considerable amount of overhead in managing(scheduling) it's processors. This is also obvious in Figure 3 where we compare *pfork* to *Mathematics of Arrays* scheduling.

From Figures 2,3 and 4, we see that scheduling algorithms using *A Mathematics of Arrays* run faster on the Alliant when we compare Alliant's 4 - 5 MIP processors to Encore's 10 - 2 MIP processors. The Alliant took a fraction of a microsecond while the Encore took approximately 1.4 milliseconds for the same operations.

On both machines, integer and floating point operations took about the same time for an outerproduct with 400 components in each of its operands. In fact, using vector registers instead of scalar registers did not make any difference except when we used Alliant's concurrency operations in C, (concurrent outer scalar inner(COSI) and concurrent outer vector inner(COVI) see Figure 2,3). On the Encore, when we forked 18 times, we not only noticed a degradation(see Figure 4) in performance, we also noticed that floating point operations ran slightly slower than scalar operations.

On the Alliant, running the CE's attached significantly effected the performance of COSI and COVI and slightly effected the performance of compiled Fortran(see Figure 3). We note that the slight speed-up we observed in the Fortan-only test versus Fortan-calling-C, is due to the overhead incurred by program linkages(see Figure 3). On the Alliant, scheduling based on *A Mathematics of Arrays* outperformed all forms of concurrency operations in either detached or attached CE mode including compiled Fortan optimized for concurrency and vectorization(see Figure 2,3).

Similary, our scheduling of array operations outperformed all concurrency instructions on the Encore. Pfork was the slowest concurrency instruction used(see Figure 4).

Conclusions and Future Research

Our experiments indicate that array operations and processor scheduling designs based on *A Mathematics of Arrays* are better than any concurrency operations used on the Alliant or Encore.

We also showed that our scheduling algorithm was true because of *Proposition 0*. We showed how a specific variety of looping program is equivalent to $<W>\rho\ \iota R$. This was formalized in *Conjecture 0* and based on *Proposition 0*.

Thus, our designs are optimal because:

(1) - They are the fastest

(2) - They are based on *A Mathematics of Arrays* and are thus theoretically sound.

(3) - They use a basic "C" and Unix instruction *fork*, and shared memory allocation. This simplicity makes our designs portable from one Parallel architecture to another. Our designs eliminate the dependencies imposed by specific Parallel architectures.

Our next report will show how the design and implementation of inner product follows from the design and scheduling of the outer product operation. Our next report will indicate why the class of structuring, partitioning, and orienting operations relate to one another. We also hope to run our experiments on other parallel architectures.

References

[1] Abrams, P. S., An APL Machine ,TR SLAC-114 UC-32(MISC), Stanford Linear Accelerator Center, February (1970).

____________, "What's Wrong with APL", In APL 75 ,pp 1-8, ACM, June, (1975).

[2] Alliant FX/Series Product Summary, Alliant Computer Systems Corp., Littleton, Mass., (1987).

[3] Backus, J., "Can Programming be Liberated from the von Neumann Style? A functional Style and Its Algebra of Programs", CACM ,V21, N8, pp613-641, (1978).

[4] Cayley, A., The Theory of Linear Transformations. An Elementary Treatise on Elliptic Functions ,pp164-178, First Edition(1876) Cambridge, Second Edition(1895) George Bell and Sons, and Dover Publications, New York (1961).

[5] Cvetanovic, Z., "The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems", IEEE Transactions on Computers, C-36, 4, pp 421-432, April (1987).

[6] Digricoli, V. J., and Harrison, M. C., "Equality-Based Binary Resolution", JACM ,V33, N2, pp253-289, (1986).

[7] Encore Computer Corp., Multimax Technical Summary, Marlboro, MA, (1987).

____________________, UMAX 4.2 Programmer's Reference Manual.

[8] Guibas, L. J., and Wyatt, D. K., "Compilation and Delayed Evaluation in APL", In Conference Record of the Fifth Annual ACM Symposium on the Principles of Programming Languages, pp 1-8, ACM, January (1978).

[9] Iverson, K. E., A Programming Language ,John Wiley and Sons, New York, (1962).

[10] Lathwell, R. H., A Formal Description of APL, November 1971, IBM Technical Report No. 320-3008.

_______________, "System Formulation and APL Shared Variables", IBM Journal of Research and Development, V17, N4, (1973).

[11] Miller, T., Tentative Compilation - A Design for an APL Compiler, TR No. 133, Department of Computer Science, Yale University, May (1978) and APL Quote Quad 9, pp 88-95, June (1979).

[12] Minter, C., A Machine Design for the Efficient Implementation of APL ,TR No. 81, Department of Computer Science, Yale University, May (1976).

[13] Mullin, L. R., Stormon, C. D., and Krempel, H. B., Initial Experiments in Parallelizing APL, CASE Report No. 8712, (1987).

[14] Multitasking User Guide (SN0222), Cray Research Inc., Mendota Heights, Minnesota.

[15] Perlis, A. J., Steps toward an APL Compiler - Updated, TR No. 24, Department of Computer Science, Yale University, March (1975).

[16] Peterson and Silberschatz, Operating Systems Concepts, Addison Wesley (1985).

[17] Reynolds, J. C., "Reasoning About Arrays", CACM ,V22, pp290-298, (1979).

[18] Robinson, J. A., "A Machine-Oriented Logic Based on the Resolution Principle", JACM, V12, pp23-41. (1965).

[19] Sylvester, J. J., "On the Theory of Partitions", Comptes Rendus ,XCVI. pp674-675, (1883) reprinted in Mathematical Papers, 5., p92, Chelsea Publishing, New York, (1973).

_______________, "Lectures on the Principles of Universal Algebra", American Journal of Mathematics ,VI, pp 270-286, (1884) reprinted in Mathematical Papers, 31., pp208-224.

_______________, "A Constructive Theory of Partitions...", American Journal of Mathematics ,V, pp 251-230, (1882); VI, pp 334-336, (1884) reprinted in Mathematical Papers, 1. pp1-83.

[20] Tu, H., FAC: A Functional Array Calculator and it's Applicaton to APL and Functional Programming, PhD Dissertation, Yale University, New Haven, Conn., (1985).

[21] Tu, H., and Perlis, A.J., "FAC: A Functional APL Language", IEEE Software ,January (1986).

[22] Whitehead,A. N., A Treatise on Universal Algebra with Applications, Hafner Publishing, New York, (1960).

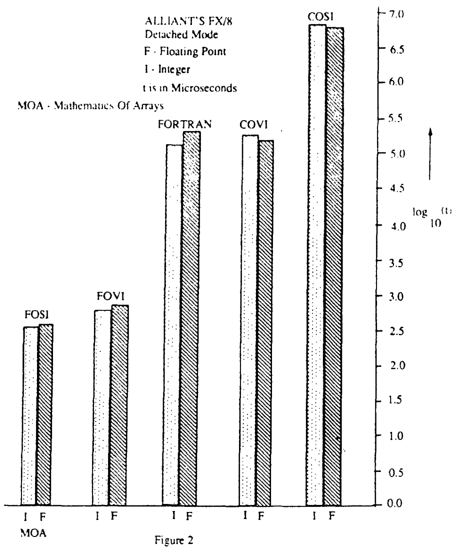


Figure 2

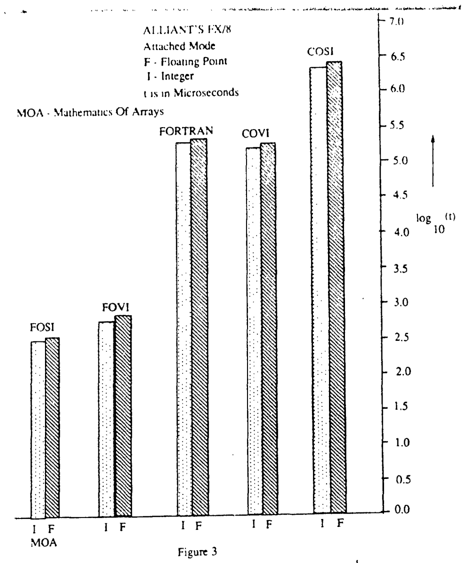


Figure 3

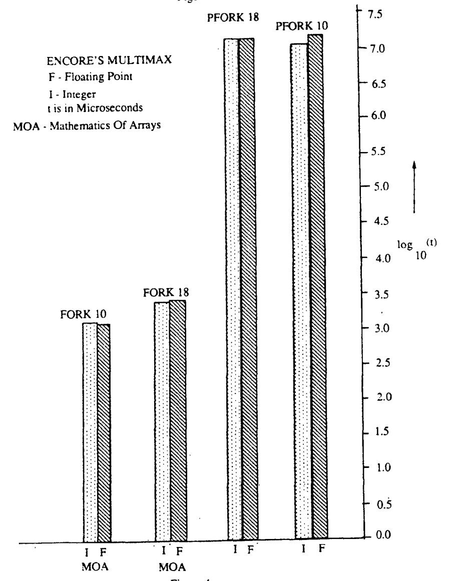


Figure 4

✓

IVY: A Shared Virtual Memory System for Parallel Computing

Kai Li
Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

A *shared virtual memory* system can provide a virtual address space shared among all processors in a loosely-coupled multiprocessor. This paper shows that such a memory can solve many problems in message passing systems on loosely-coupled multiprocessors, and describes the design and implementation of a prototype shared virtual memory system, IVY, implemented on an Apollo ring network. The experiments on the prototype system show that parallel programs using a shared virtual memory yield almost linear and occasionally super-linear speedups and that it is practical to implement such a system on existing architectures.

Introduction

Much of the work on distributed computing has focused on message passing models such as Hoare's communicating sequential processes [16] and Actor [15], perhaps because message passing matches the basic communication mechanism in loosely-coupled multiprocessors. Many people have studied shared memory models for tightly coupled multiprocessors, but few have studied that model for loosely-coupled multiprocessors. Because not enough work has been done, it has not been clear whether a message passing model is better than a shared memory model for parallel computation on loosely-coupled multiprocessors. It has also not been clear whether it is possible to design an efficient system to support the shared memory model on loosely-coupled multiprocessors.

Systems based on message passing suffer mainly in two aspects: passing complex data structures and process migrations. This paper shows that a solution to these problems is to build a *shared virtual memory* . The shared virtual memory provides a virtual address space that is shared among all processors in a loosely-coupled distributed-memory multiprocessor system. Application programs can use the shared virtual memory just as they do a traditional virtual memory, except that processes can run on different processors in parallel. The shared virtual memory keeps its memory pages coherent all the time and data can naturally *migrate* between processors on demand [23,22]. Furthermore, just as a conventional virtual memory swaps *processes*, so does the shared virtual memory. Thus the shared virtual memory provides a natural and efficient form of *process migration* between processors in a distributed system. This is quite a gain because process migration is usually very difficult to implement. In effect, process migration subsumes *remote procedure calls*.

A prototype shared virtual memory has been implemented on a network of Apollo workstations. A number of practical parallel program examples are chosen to run on the prototype system. The experimental results show that parallel programs using such a not well-tuned, user-mode shared virtual memory system yield almost linear and occasionally super-linear speedups over a uniprocessor. The success of this implementation suggests a new operating mode for loosely-coupled multiprocessor architectures in which parallel programs can exploit the total processing power and memory capabilities in a far more unified way than the traditional "message-passing" approach.

Shared Memory vs. Message Passing

Message passing in concurrent systems is characterized by multiple threads of control. A pure message passing system usually does not have any shared global data; instead processes access ports or mailboxes to achieve interprocess communication. Parallel programs need to use primitives such as *send* and *receive* explicitly through channels, ports, or mailboxes. Although programmers can use these primitives to synchronize parallel programs, they need to be conscious of data movement between processes at all times.

Remote procedure call is a mechanism for language-level transfer of control and data between programs in disjoint address spaces whose primary communication medium is a narrow channel [24]. A remote procedure call mechanism allows programmers to worry less about data movement and provides clients with a fairly transparent interface so that remote procedure calls look much like local procedure calls. However, the transparency of remote procedure calls is limited because a remote procedure call mechanism actually simulates the execution in the same address space using completely different address spaces.

Since both message passing and remote procedure calls deal with multiple address spaces, they both have difficulties with passing complex data structures. In fact, the difficulty of passing complex data structures is the main drawback of message passing and remote procedure calls for parallel programming. For example, passing a list data structure by sending messages will introduce considerable complexity in programming and substantial overhead in both space and time [14]. In a remote procedure call, there is no good way to pass a pointer argument [24]. This problem becomes more severe when the data structures are fundamental to a language being implemented on a parallel machine.

In contrast, a shared memory multiprocessor has no difficulty passing pointers because processors can share a single address space. Therefore, there is no need to pack and unpack the data structures containing pointers in messages. Passing a list data structure simply requires passing a pointer.

Another problem with message passing systems is the difficulty of process migration because there are multiple address spaces. When migrating a process, all the operating system resources allocated by the process have to be moved together; this is expensive [25]. In the case where a process has a few opened ports and files, the pending messages and file access control blocks need to be transferred. Furthermore, the code and the stack of the process have to be moved because there is no easy way to translate the contents of different address spaces efficiently on the fly.

In a shared memory multiprocessor system, a process migration only requires moving a process from the ready queue on the source processor to the ready queue on the destination processor because process control block, code, and stack are all in the same address space.

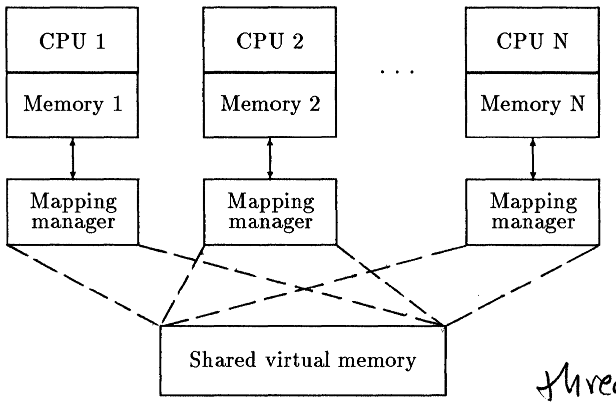
Some systems use a set of primitives to access a global space that is used to store shared data structures of processes [8,5]. Although programming the global space does not require data movement as much as message passing, programmers still have to explicitly use the primitives. In a primitive global-space system, passing complex data structures and process migration are as difficult as in message passing systems, since accessing the data structures and process migration are by value or by name. Furthermore, using primitives may greatly reduce the efficiency of parallel programs because a primitive operation requires at least one procedure call, which costs much more than a simple memory reference.

Both data structure passing and process migration are important for implementing parallel programming languages. Although some implementations of parallel programming languages are based on a message passing facility, implementing existing parallel languages on a shared memory multiprocessor can greatly simplify the implementations. In summary, shared memory is highly desirable for parallel computation.

Shared Virtual Memory

A *shared virtual memory* is a single address space shared by a number of processors (Figure 1). Any processor can access any memory location in the address space directly. Memory mapping managers implement the mapping between local memories and the shared virtual memory address space. Other than mapping, their chief responsibility is to keep the address space *coherent* at all times; that is, the value returned by a read operation is always the same as the value written by the most recent write operation to the same address. In short, a shared virtual memory provides clients with the same interface as the shared memory address space on a shared-memory multiprocessor.

A shared virtual memory address space is partitioned into *pages*. Pages that are marked *read-only* can have copies residing in the physical memories of many processors at the same time. But a page marked *write* can reside in only one processor's physical memory. The memory mapping manager views its local memory as a large cache of the shared virtual memory address space for its associated processor. Like traditional virtual memory [11], the shared memory itself exists only *virtually*. A memory reference will cause a page fault when the page containing the memory location is not in a processor's current physical memory. When this happens, the memory mapping manager retrieves the page from either disk or the memory of another processor. If the faulting memory reference is the target of a write operation, then the memory mapping manager must guarantee the atomicity of the operation [23].



threads

Figure 1: Shared virtual memory mapping

In a shared virtual memory system, the model of a parallel program is a set of *processes* (or threads) that share a single virtual memory address space. These processes are "lightweight"—they share the same address space and thus the cost of a process context switch, process creation, or process termination is small, say, on the order of a few procedure calls [20]. One of the key goals of the shared virtual memory, of course, is to allow processes of a program to execute on different processors in parallel. To do so, the appropriate process manager and memory allocation manager must be integrated properly with the memory mapping manager.

The performance of parallel programs on a shared virtual memory system depends mainly on two things: the number of parallel processes and the degree of data sharing (i.e. contention). Theoretically, performance improves as the number of parallel processes increases and contention decreases. Contention is less if a program exhibits *locality of references*. One of the main justifications for the traditional virtual memory is that memory references in sequential programs generally exhibit a high degree of locality [10,12]. Although memory references in parallel programs may behave differently from those in sequential ones, a single process is still a sequential program, and should exhibit a high degree of locality. Contention among parallel processes for the same piece of data depends on the algorithm, of course, but a common goal in designing parallel algorithms is to minimize such contention for optimal performance.

Prototype Implementation

In order to answer the question of whether it is practical to build a shared virtual memory on a loosely-coupled multiprocessor and whether most parallel application programs will get speedup on such a system, a user-mode prototype system has been implemented on the Apollo Domain [1,21], an integrated system of personal workstations and server computers connected by a 12M bit/sec baseband, single token ring network. IVY is implemented on top of the modified operating system Aegis of the Domain environment. The implementation is not particularly efficient but simple and tractable.

IVY consists of 5 modules, namely, remote operation, memory mapping, process management, memory allocation, and initialization. The hierarchy of the system is shown in Figure 2. The three top modules in the hierarchy form the IVY client interface. Each consists of a set of primitives that can be used by application programs.

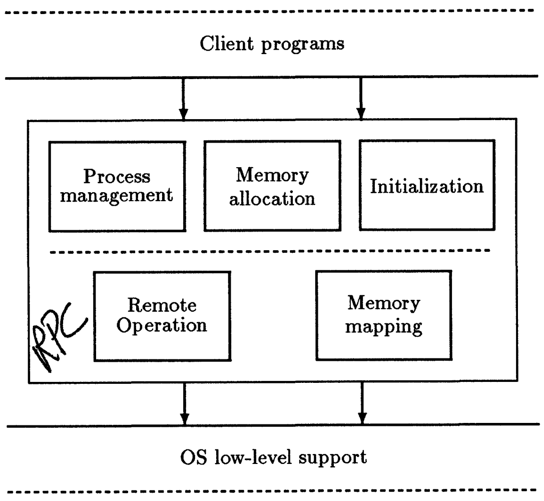


Figure 2: IVY hierarchy.

Shared Virtual Memory Mapping

Memory mapping managers implement the mapping between local memories and the shared virtual memory address space. Other than mapping, their chief responsibility is to keep the address space *coherent* at all times; that is, the value returned by a read operation is always the same as the value written by the most recent write operation to the same address. The memory coherence problem is similar to that encountered in cache and multicache designs for shared memories on multiprocessors (see [27,2] for a survey), but most memory coherence techniques for multicaches do not apply, because a loosely-coupled multiprocessor has no physically shared memory and the communication cost between processors is non-trivial. [23] gives a detailed description and analysis of the algorithms for memory coherence.

Since memory coherence memory coherence of a shared virtual memory is maintained at page level, it is important to choose the right page size. On a stock loosely-coupled multiprocessor, one has to use a page size which is consistent with or the multiple of that provided in a Memory Management Unit (MMU) in order to use its protection mechanisms to detect incoherent memory references and trap them to appropriate fault handlers. These page fault handlers and their servers implement memory coherence strategy that keeps the memory space coherent at all times. Since sending large packets of data (say 1,000 bytes) in a loosely-coupled multiprocessor is not much more expensive than sending small ones (say 100 bytes) [28], relatively large page sizes are possible in a shared virtual memory. On the other hand, the larger the memory unit, the greater the chance for contention. The possibility of contention indicates the need for relatively small page sizes. Our experience with a page size of 1K bytes has been pleasant and we expect that smaller page sizes (perhaps as low as 256 bytes) will work well also, but we are not as confident about larger page sizes, due to the contention problem. The right size is clearly application dependent, however, and we simply do not have the implementation experience to say what size is best for a sufficiently broad range of parallel programs.

In IVY, each user address space is divided into two portions. The shared virtual memory address space is in the high portion and the private memory is in the low portion. For simplicity, the data structure of the page table is a vector of records and each record is a table entry. The whole table is stored in the private memory.

The memory coherence strategies implemented IVY use *invalidation* approach. In this approach, all read-only copies of a page are invalidated (changed to nil access) before a processor writes to a page. For experimental purposes, we implemented three algorithms: the improved centralized manager algorithm, the fixed distributed manager algorithm, and the dynamic distributed manager algorithm. These algorithms and other algorithms for solving the memory coherence problem have been studied in depth [23]. Briefly, The *centralized manager* algorithm is similar to the cache coherence solution [6]. The centralized manager resides on a single processor, and maintains all ownership information. When having a page fault, a processor will ask the manager for the copy of the page. The manager will then ask the owner of the page to send a copy to the requesting processor.

The *fixed distributed manager* algorithm gives every processor a predetermined set of pages to manage. The most straightforward approach is to distribute pages evenly in a fixed manner to all processors (the distributed directory map solution to the multicache coherence problem [2] is similar). With this approach there is one manager per processor, each responsible for the pages specified by the fixed mapping function H. When a fault occurs on page p, the faulting processor asks processor $H(p)$ where the true page owner is, and then proceeds as in the centralized manager algorithm.

The *dynamic distributed manager algorithm* keeps track of the ownership of all pages in each processor's local page table, using a field called *probOwner* in each page entry. The value of this field can be either the true owner or the "probable" owner of the page. The information that it contains is just a hint; it is not necessarily correct at all times, but if incorrect it will at least provide the beginning of a sequence of processors through which the true owner can be found. Initially, the *probOwner* field of every entry on all processors is set to some default processor that can be considered the initial owner of all pages. As the system runs, each processor uses the *probOwner* field to keep track of the last change of the ownership of a page. This field is updated whenever a processor receives an invalidation request, relinquishes ownership of the page, or forwards a page fault request.

The fixed distributed manager algorithm, the dynamic distributed manager algorithm, and their variations are more appropriate than others.

Process and Process Scheduling

The process management module implements all the operations for process control, process migration, and process synchronization. The module provides clients with a set of calls for writing parallel programs.

All the processes in IVY are lightweight. The program code of a process is stored in its private memory; therefore, IVY need not have its own loader. The stack of a process is allocated from the shared memory portion. Each process has a process control block (PCB) that contains necessary information like process state, stack, context, and other process control-dependent information. The PCBs are stored in the private memory of the address space. Therefore, the PID of a process is represented as a pair—processor number and the address of its PCB.

The process scheduling mechanism is designed to be simple. Each processor has a local ready queue using a last-in-first-out policy, that is, processes do not have priorities. The process dispatcher always picks up the process in the front of the ready queue. If there is no ready process available, the dispatcher runs a system process called the null process.

The null process implements a passive load balancing algorithm. It normally waits on two low level eventcounts, one for timeout and another for new ready processes. The null process is invoked when either of them is advanced. When a timeout event occurs, the null process will run the passive load balancing algorithm. The main idea of the algorithm is to let each processor ask for work when it is idle using some hints. The eventcount for new ready processes can be advanced only when a process is migrated to the current processor, a remote resume operation is performed, or a remote notification operation results in waking up a process. Of course, when a new ready process is available, the null process will suspend itself. The dispatcher will then do another schedule.

The hint information about the number of ready processes is important for minimizing the number of rejections of migration requests. The processors in IVY keep each other up to date on their current work loads by adding a few extra bits to the messages transmitted for remote operations. Usually, a byte will be enough to transfer the information. This byte can be packed into every message at almost no extra cost.

Experiments with many parallel application programs show that the algorithm will not work well if the number of ready processes on each processor is used as the only criterion for migrating processes. A better way is to use the number of processes (including both ready and suspended) controlled by thresholds [22]. When such a number is less than the lower threshold, the processor will try to ask for work. When such a number is greater than the upper threshold, the processor will migrate processes to other processors upon requests.

Process migration

A process in IVY is either migratable or non-migratable, indicated by a field in its PCB. Clients can modify the field by using a primitive so that a migratable process can become non-migratable or vice versa at run time. Only a ready, migratable process can migrate from one processor to another. When a process is migrated, a forwarding pointer is put into its PCB and the migrated attribute is set. The PCBs of migrated processes are used for storing forwarding pointers. The collection of non-reachable PCB's has not been implemented in IVY.

Since PCBs are stored in the private memory portion of the address space, a process migration must

- send the PCB of the process to the destination processor and put it into a PCB,
- copy the current page of the process's stack to the destination processor and transfer the ownership of the page,
- transfer the ownership of all the pages in the upper portion of the stack to the destination processor, and
- put the PCB in the ready queue on the destination processor.

The reason for moving the current page of the process's stack is to avoid a page fault in the process dispatcher (Figure 3).

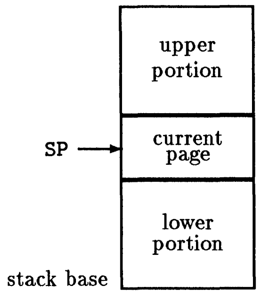


Figure 3: A process stack.

The upper portion of the stack need not move to the destination processor because its content is meaningless. Ownership transfer is inexpensive because it only requires setting the protection bits of the page frames. There is no need to do anything with the lower portion of the stack because the stack can grow without having further page faults after the current page and the upper portion of the stack become writable.

Eventcount Implementation

In a shared virtual memory system, it is possible to implement a process synchronization mechanism based on either global memory or message passing. Eventcount [26] is the process synchronization mechanism in IVY. The main reason for choosing eventcount is that the Aegis operating system uses eventcounts as its synchronization mechanism.

An eventcount synchronization mechanism has four primitive operations:

- `Init( ec )` — initializes an eventcount.
- `Read( ec )` — returns the value of the eventcount.
- `Await( ec, value )` — suspends the calling process itself until the value of the eventcount reaches the value specified.
- `Advance( ec )` — increments the value of the eventcount by one and wakes up awaiting processes.

After an eventcount is initialized, any process can use it without knowing where it resides.

The implementation of these primitives is based on shared virtual memory. The atomic operation is implemented by pinning memory pages and using test-and-set instructions. This implementation is much cleaner than that based on message-passing; furthermore, the performance is better when there is more than one process on each processor because eventcount primitives become local operations when the eventcount data structure has been paged into the local processor.

The data structures of an eventcount usually reside together in one page. The shared virtual memory mapping mechanism can move this page on demand when an eventcount operation is performed and on a processor where there is no such eventcount data structures. If the data structures of an eventcount require more than one page, then the additional pages will be linked together. This mechanism increases the locality of the eventcount data structure. In most cases, only one page is needed for each eventcount.

Memory Allocation

IVY has a simple memory allocation module that uses a "first fit" algorithm with one-level centralized control. The processor with which the user directly contacts will be appointed to the centralized memory manager. To reduce the memory contention, the memory allocator allocates each piece of memory to the boundary of a page.

Both allocate and free are atomic operations. IVY uses a binary lock on each processor for memory allocation purposes. At the beginning of each memory management primitive, a test-and-set operation is performed on the lock. A failed process will be put into a queue and will be awakened by an unlock operation on the lock which is done at every end of each primitive.

A more efficient approach is two-level memory management. In this approach, each processor has a local allocator maintaining a big chunk of memory allocated from the central memory allocator. This big chunk of memory serves for the local memory allocations. When there is not enough free memory left in the big chunk, the local allocator will allocate another big chunk from the central allocator. This approach has not been implemented yet, though it is expected to have better performance.

Remote Operation

The remote operation module implements a remote request/reply mechanism that handles all the remote operations of other modules. Such a mechanism (also called simple RPC) is similar to remote procedure call facility [24,3], but it is simpler than the general one and has a few special features for implementing shared virtual memory system.

One of such features is broadcast or multicast remote operation mechanism. A broadcast or a multicast request has three reply schemes: a reply from any receiving processor, replies from all receiving processors, and no reply at all. The first option is useful for broadcasting page fault requests to locate page owners (see [23]). The second option can be used for implementing invalidation operations. The third option is for broadcasting approximate information for process scheduling.

Another feature is a forwarding request mechanism that allows a processor to forward a request to another processor. For example, processor 1 can send a request to processor 2, processor 2 forwards the request to processor 3, and so on until processor k performs the operation and sends a reply back to processor 1. There are no intermediate replies involved in the operation. This mechanism is particularly useful for implementing the dynamic distributed manager algorithm.

The retransmission protocol is based on the philosophy of resending replies only when necessary. Such a design is based on two assumptions: local computation is always correct, and communication may be unreliable, but once a packet is received, its content is always correct. The protocol is reliable only when these assumptions hold. In practice, the assumptions are reasonable. Retransmission checking is done in a null process, which checks all the outgoing channels every half second when there is nothing to do.

Programming in the IVY Environment

Programmers can use any programming language in the Apollo DOMAIN to write parallel programs as long as they can interact with the procedure calls in the Apollo DOMAIN Pascal in which IVY is implemented. Since all the languages in the Apollo DOMAIN are designed for sequential programming, the programmer has to program parallel constructs explicitly with the primitives provided by IVY.

Programmers or compilers using IVY need to decide which piece of data puts into shared virtual memory and which into private memory. Programs later do not need to know where the shared data structures are in the sense that references to these data structures are the same as to other data structures. If IVY had its own loader, explicit memory allocation would not be necessary.

Clients can use primitives provided by the process management module to create lightweight processes (or threads) for a parallel computation. The programmer can choose how to schedule processes when calling an initialization procedure at the beginning of the program. There are two options: manual scheduling and system scheduling. If system scheduling is used, the programmer only needs to create and terminate processes. But if manual scheduling is chosen, the programmer needs to tell where and when a process goes. It is the programmer's responsibility to program process synchronization. The methodology of such programming is the same as that of "conventional" concurrent programming developed since the 1960s. Although there is no parallel programming language, such a primitive environment has proven to be convenient enough to write benchmark programs.

IVY does not have any special debugging tools. Initial debugging programs can be done on a single processor. Since an IVY image file can run on any number of processors, there is no need to have a simulator. If a program follows IVY parallel programming conventions, debugging on a single processor is usually easy. After debugging on a single processor, the programmer should debug on two and then three processors. My experience indicates that if a program can run on three processors correctly, there are few bugs left.

Experiments

Given the difficulties of finding practical parallel programs, the only reasonable way to do experiments is to select a set of application programs from different fields as a benchmark suite. All benchmarks have the following two properties:

- *reasonably fine granularity of parallelism, and*
- *side-effects in shared data structures.*

Parallel programs with rather coarse granularity can obviously perform well in the shared virtual memory system. There are parallel functional programs that do not have any side-effects in their data structures at run time. The shared virtual memory system is clearly a big win in these applications. The main goal in using the two criteria is to avoid weighing the experiments in favor of the shared virtual memory system by picking problems that suit the system well. The benchmark set in the experiments consists of six parallel programs that are written in Pascal. All of them are transformed manually from sequential algorithms into parallel ones in a straightforward way.

Linear Equation Solver This program implements a parallel Jacobi algorithm for solving linear equations. The algorithm is transformed from the traditional, sequential Jacobi algorithm that solves the linear equation $Ax = b$ where A is an n by n matrix. In each iteration, $x^{(k+1)}$ is obtained by

$$x_i^{(k+1)} = \Big(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\Big)/a_{ii}.$$

The parallel algorithm creates a number of processes to partition the problem by the number of rows of matrix A. All the processes are synchronized at each iteration by using an eventcount. The data structures A, x, and b are stored linearly in the shared virtual memory, and the processes access them freely without regard to their location.

3D PDE Solver This program solves three dimensional partial differential equations (PDEs) using a parallel Jacobi algorithm. The algorithm and its transformation are similar to the linear equation solver except that in the equation $Ax = b$, A is a sparse matrix. Since this matrix is never updated in the program, the practical PDE solvers in scientific computing usually eliminate the matrix by coding it into programs to save space and time. In practice, matrix A is large and it is read-only, coding it into program will not be in favor of the shared virtual memory performance. To be more realistic, we choose to do so. The vectors x and b are stored linearly in the shared virtual memory.

Traveling Salesman Problem The traveling salesman problem is to find a tour that visits each city once with the minimum cost. The cities are represented as the nodes in an undirected graph. The cost of a tour is the sum of the weights of the edges on the tour. The algorithm used in the program is a simplified version of the branch-and-bound approach proposed in [13]. At each step, an 1-tree (a variation of the minimum spanning tree) of the remaining graph is computed. The sum of the cost of the subtour and the 1-tree is compared with the cost of the current least upper bound. If the cost is less than the upper bound, it will replace the upper bound and the subtour is still valid; otherwise, the subtour will be thrown away. The available branches, the graph, and the least upper bound are stored in the shared virtual memory. The program creates a process for each processor that performs the branch-and-bound algorithm on a branch obtained from the shared virtual memory. These processes run in parallel until the tour is found. Each process is not much different from the sequential one except it needs to access shared data structures mutual exclusively.

Matrix Multiply This program computes $C = AB$ where A, B and C are square matrices. A number of processes are created to partition the problem by the number of columns of matrix B. All the matrices are stored in the shared virtual memory. The program assumes that matrix A and B are on one processor at the beginning and they will be paged to other processors on demand.

Dot-product The dot-product program computes

$$S = \sum_{i=1}^{n} x_i y_i.$$

A number of processes are created to partition the problem. Each process computes a partial sum and S is obtained by summing up the partial sums produced by the individual processes. Both vector x and y are stored in the shared virtual memory in a random manner, under the assumption that x and y are not fully distributed before doing the computation. The main reason for choosing this example is to show the weak side of the shared virtual memory system; dot-product does little computation but requires a lot of data movement.

Split-merge Sort This program implements a variation of the block odd-even based merge-split algorithm described in [4]. The sorted data is a vector of records that contain random strings. At the beginning, the program divides the vector into $2N$ blocks for N processors, and creates N processes, one for each processor. Each process sorts two blocks by using a quicksort algorithm [17]. This internal sorting is naturally done in parallel. Each process then does an odd-even block merge-split sort $2N-1$ times. The vector is stored in the shared virtual memory, and the spawned processes access it freely. Because the data movement is implicit, the parallel transformation is straightforward.

The speedup of a program is the ratio of the execution time of the program on a single processor to that on the shared virtual memory system. In order to obtain a fair speedup measurement, all the programs in the experiments partition their problems by creating a certain number of processes according to the number of processors used. As a result of such a parameterized partitioning, any program does its best for any given number of processors. Unlike message-passing systems or primitive global-space systems, IVY has almost no extra overhead when programs run on a single processor. The only additional costs are in creating processes, which takes milliseconds in total, and mutual exclusion, which takes two 68000 instructions for each locking. Since there are few locking operations in the programs above, the programs using one processor run just as fast as their sequential programs.

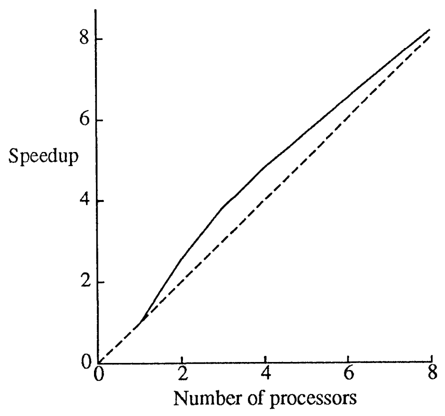


Figure 4: Super-linear speedup

The 3D PDE program, when matrix A is 50^3 by 50^3, experienced super-linear speedup as shown in Figure 4. At first glance, the result seems impossible because the fundamental law of parallel computation says that a parallel solution utilizing p processors can improve the best sequential solution by at most a factor of p. Since the algorithm in both programs is a straightforward transformation from the sequential Jacobi algorithm and all the processes are synchronized at each iteration, the algorithm cannot yield super-linear speedup. The reason is that the fundamental law of parallel computation assumes that every processor has an infinitely large memory, which is not true in practice. For instance, in the parallel 3-D PDE example, the data structure for the problem is greater than the size of physical memory on a single processor, so when the program is run on one processor there is a large amount of paging between the physical memory and disk.

Table 1 shows the total number of disk I/O page transfers of the first six iterations when the 3D PDE program runs on one processor and two processors. Obviously, the number of the disk I/O page transfers on two processors is substantially less than that on one processor. In the two-processor case, the program initializes its data structures only on one processor, this processor causes most disk I/O transfers because it cannot hold all the data structures in its physical memory. As the program runs, the processes start to access some portions of the data structures, causing the shared virtual memory page faults to move pages from one processor to another. When the shared virtual memory distributes the data structure into individual physical memories whose cumulative size is large enough, few disk I/O data movements will occur. On the other hand, IVY is a user-mode system implemented on top of the Aegis virtual memory system which performs an approximate LRU page replacement strategy; the pages recently moved from the processor with initialized data structures may not be replaced because these pages are also most recently referenced ones. This explains why the number of disk I/O page transfers in the two-processor case decreases gradually.

	Disk page transfers of each iteration					
	1	2	3	4	5	6
1 processor	899	1600	1543	1515	1542	1540
2 processors	1432	1072	466	156	101	105

Table 1: Disk page transfers

When the data structure of the problem is not larger than the physical memory on a processor (matrix A is 40^3 by 40^3), the result of the 3D PDE is no longer super-linear, as shown in Figure 5. They are similar to what we see in the past. For example, the result is similar to that generated by similar experiments on CM*, a shared memory multiprocessor [18,9]. Indeed, the shared virtual memory system is as good as the best curve in the published experiments on CM* for the same program; but the efforts and costs of the two approaches are dramatically different. In fact, the best curve in CM* was obtained by keeping the private program code and stack in the local memory on each processor. The main reason that the performance of this program is so good in the shared virtual memory system is that the program exhibits a high degree of locality. While the shared virtual memory system pays the cost of local memory references, CM* pays the cost of remote memory references across its Kmaps.

The dot-product program did not perform well on IVY, as indicated in Figure 5. It is included here so as not to paint too bright a picture. Since this program only references each element once, the ratio of the communication cost to the computation cost in this program is large. For programs like dot-product, it is not appropriate to use a shared virtual memory system, unless the communication cost can be reduced.

Matrix multiply and traveling salesman problem perform well on IVY system. They show the good side of the shared virtual memory system. Both programs exhibit a high degree of localized computation. Since the algorithm used in the traveling salesman problem is a parallel branch-and-bound, there are

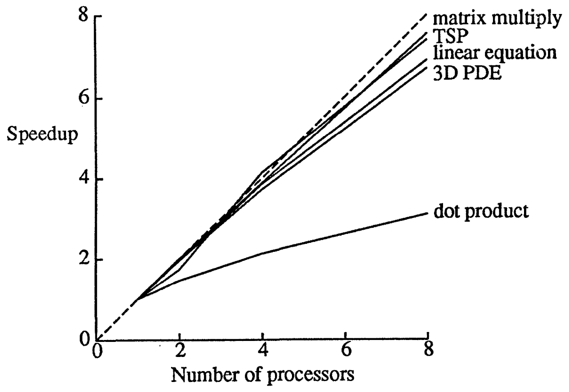


Figure 5: Speedups

anomalies [19]. It is possible that the program gets super-linear speedup or no speedup at all. In this example, it happens to have super-linear speedup.

Figure 6 shows the speedup of merge-split sort program. The curve does not look very good because even with no communication costs, the algorithm does not yield linear speedup. The program uses the best strategy for any given number of processors. For example, there is one merge-split sorting when running the program on one processor, there are 4 blocks when running the program on two processors, and so on. Using a fixed number of blocks for any number of processors would result in a better speedup, but such an approach is not reasonable.

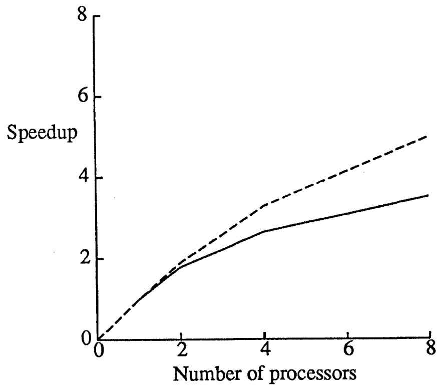


Figure 6: Speedup of merge-split sort

Conclusion

The difficulties with passing complex data structures and process migration are the main drawbacks of the message passing model for parallel computing. Shared virtual memory on loosely-coupled multiprocessors can solve these problems. The success of implementing the prototype shared virtual memory system IVY and the experiments show that it is practical to implement such a system on existing loosely-coupled multiprocessors such as local area networks.

The implementation experience shows that, although it is possible to implement a shared virtual memory without modifying an existing system like the Aegis operating system, it is necessary to modify the existing system to get acceptable performance. IVY is a user-mode implementation, so it has a lot of overhead. A system-mode implementation ought to provide a substantial improvement. It is expected that a well-tuned system-mode implementation should improve the performance of remote operations and page moving by a factor of at least two according to the performance comparison with some well tuned systems [28,7]. I/O overlaps among the lightweight processes do not exist in IVY. An integrated heavyweight and lightweight process scheduler is highly desirable. The disk I/O overlap may also greatly improve IVY's performance.

The experimental results of running some non-trivial parallel programs on the prototype system strongly support the idea of shared virtual memory on loosely-coupled multiprocessors. The results demonstrate that the shared virtual memory can effectively exploit not only the available processors but also the combined physical memories of a multiprocessor system.

The experimental results reported in this paper are limited because there were only up to eight processors available for running the modified Aegis operating system at the time. Experiments on more processors will show more insights of shared virtual memory and behaviors of parallel programs. To answer many unanswered questions, we plan to perform more experiments on a shared virtual memory system being implemented on a large-scale multiprocessor at Princeton.

Acknowledgement

I wish to thank John Ellis for his invaluable suggestions and helpful discussions and Nat Mishkin for his help with understanding the Aegis kernel, which made my modifications to the OS possible. I would like to thank Paul Hudak and Alan Perlis for their continual help inspiration. I also wish to thank Jeff Naughton and the referees for their helpful comments.

References

[1] Apollo. *Apollo DOMAIN Architecture.* Apollo Computer Inc., Chelmsford, Mass., 1981.

[2] J. Archibald and J. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[3] A.D. Birrell and B.J. Nelson. *Implementing Remote Procedure Calls.* Technical Report CSL-83-7, Xerox PARC, December 1983.

[4] D. Bitton, D.J. DeWitt, D.K. Hsaio, and J. Menon. A Taxonomy of Parallel Sorting. *ACM Computing Surveys*, 16(3):287–318, September 1984.

[5] N. Carriero and D. Gelernter. The S/Net's Linda Kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[6] L.M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[7] David R. Cheriton. The V kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19–43, 1984.

[8] D.R. Cheriton and M. Stumm. The Multi-Satellite Star: Structuring Parallel Computations for A Workstation Cluster. To appear, 1988.

[9] Jarek Deminet. Experience with Multiprocessor Algorithms. *IEEE Transactions on Computers*, C-31(4), April 1982.

[10] Peter J. Denning. On Modeling Program Behavior. In *Proceedings of Spring Joint Computer Conference*, pages 937–944, AFIPS Press, 1972.

[11] Peter J. Denning. Virtual Memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[12] Peter J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[13] M. Heid and R.M. Karp. The Traveling-salesman Problem and Minimum Spanning Trees. *Operation Research*, 17(12):1139–1167, December 1970.

[14] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[15] Carl Hewitt. The Apiary Network Architecture for Knowledgeable Systems. In *Proceedings of the Lisp Conference*, pages 107–117, August 1980.

[16] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(11):666–677, August 1978.

[17] C.A.R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[18] A. K. Jones and P. Schwarz. Experience Using Multiprocessor Systems — A Status Report. *ACM Computing Surveys*, 12(2), June 1980.

[19] T. Lai and S. Sahni. Anomalies in Parallel Branch-and-Bound Algorithms. *Communications of the ACM*, 27(6):594–602, June 1984.

[20] B. M. Lampson and D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[21] P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson, and B.L. Stumpf. The Architecture of an Integrated Local Network. *IEEE Journal on Selected Areas in Communications*, 1983.

[22] Kai Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors.* PhD thesis, Yale University, October 1986. Tech Report YALEU-RR-492.

[23] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986. A journal version will appear in *ACM Transactions on Computer Systems.*

[24] Bruce J. Nelson. *Remote Procedure Call.* PhD thesis, Carnegie-Mellon University, May 1981.

[25] M.L. Powell and B.P. Miller. Process Migration in DEMOS/MP. In *Proceedings of the ninth Symposium on Operating Systems Principles*, pages 110–119, 1983.

[26] David P. Reed and Rajendra K. Kanodia. Synchronization with Eventcounts and Sequencers. *Communications of the ACM*, 22(2):115–123, February 1979.

[27] Alan J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[28] Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, 25(4):260–273, April 1982.

Hierarchical Workload Allocation for Distributed Systems

Nicholas S. Bowen
Christos N. Nikolaou

IBM Research Division
T.J. Watson Research Center
P.O. Box 704, H4-G51
Yorktown Heights, N.Y. 10598

Arif Ghafoor

Department of Electrical and Computer Engineering
Syracuse University
Syracuse, N.Y. 13244-1240

Abstract

Workload management is one of the important open problems in distributed operating systems research. The mapping of the distributed applications onto the processors of the system has to be achieved in such a way so that the performance goals of these applications (response time, throughput) are met. Balancing the load of the applications among the processors of the distributed system can increase the overall throughput. However, if two or more highly interacting processes are assigned to different processors, their response time may suffer because of the communication overhead.

We propose and evaluate an efficient hierarchical clustering and allocation algorithm that drastically reduces the interprocess communication cost while observing low and upper bounds of utilization for the individual processors. We compare the algorithm with branch-and-bound-type algorithms that can produce allocations with minimal communication cost, and we show a very encouraging time complexity/suboptimality trade-off in favor of our algorithm, at least for a class of process clusters and their random combinations, which we believe occur naturally in distributed applications. Our heuristic allocation is well suited for a changing environment, where processors may fail or be added to the system and where the workload patterns may change unpredictably and/or periodically.

1.0 Introduction

One of the reasons that the performance of a distributed system may suffer is the following: if the mapping of distributed applications to processors is not carefully implemented, processors may find themselves spending most of their time waiting for each other instead of performing useful computations. A fundamental trade-off is at work here: one would like to spread the load of a distributed application as "evenly" as possible among the processors of the system in order to maximize throughput. But if heavily used (or high priority) processes communicating frequently with each other, for example through messages, shared memory, remote procedure calls etc., are assigned to different processors, then significant time could be wasted executing expensive global synchronization protocols such as interprocessor locking or blocking *SENDs* and *RECEIVEs*. In addition, the interprocessor communication (locking) cost is usually significantly higher than the cost incurred when two processes communicate within the same processor. A survey of the work on the load allocation and balancing problem can be found in [Bowe87]

In this paper, we propose and evaluate a hierarchical clustering algorithm that can be used for grouping both processes and processors in clusters of either frequently communicating processes or topologically close processors. The clustering algorithm generates two cluster trees, one for the processes and one for the processors. We next evaluate a heuristic mapping algorithm, proposed in [Niko86], and we compare its results with an optimal allocation of processes to processors that minimizes the total process communication cost while keeping the workload assigned to each processor within some previously specified lower and upper bounds. We show that for a class of process clusters and their random combinations, which we believe naturally occur in distributed applications, the allocation obtained by our heuristic is close to the optimal. Our heuristic allocation is well suited for a changing environment, where processors may fail or be added to the system and where the workload patterns may change unpredictably and/or periodically. In [Niko86, Ferg86] algorithms are given to relocate processes when the configuration changes. These algorithms modify the process and processor cluster trees generated during the original allocation, to reflect the configuration changes. These changes, however, do not usually alter drastically the two trees. Therefore, only a small subtree of the process cluster tree will have to be remapped to a small processor cluster subtree.

The rest of the paper is organized as follows: we give a formal definition of the problem in 2.0, Formulation of the problem. We then present our clustering algorithm in 3.0, Clustering and we review the heuristic allocation algorithm proposed in 4.0, Allocation Algorithm. In 5.0, Results, we present the results of applying both the heuristics and integer linear programming techniques to the allocation problem.

2.0 Formulation of the problem

In the context of this paper we conceive of a distributed system as composed of a set of *nodes* representing the active processing agents (e.g. host computer complexes in a computer network, individual processors in a multiprocessor system or a local area network, etc.) and an *interconnection* structure providing full connectivity between the nodes (e.g. communication lines in a long-haul computer network, shared memory modules or a bus in a multiprocessor system, a ring or a star topology in a local area network). Given a set of processes Δ_n, a set of processors Π_n and their logical and physical interconnection structure respectively, we define the *process graph* $\Delta = (\Delta_n, \Delta_l)$, where Δ_l is a set of links defined as: $\Delta_l = \{(i, j) \mid (i, j)$ denotes logical communication between processes i and $j\}$. Similarly we define the *processor graph* $\Pi = (\Pi_n, \Pi_l)$, where $\Pi_l = \{(k, l) \mid (k, l)$ denotes a physical communication link between processors k and $l\}$.

The process allocation problem can be formulated as a quadratic assignment problem, and as such is similar to several other assignment problems such as facilities location, space allocation, scheduling and routing, [Ligg81]. These problems differ from the linear assignment problem in that the entities to be assigned are treated as a set of interconnected, rather than independent, objects (e.g. a facilities location problem in which the cost of materials flow between facilities is an important consideration). In our case, the interconnection of processes is modelled by a cost matrix A with a typical element a_{ij} , denoting the volume of communication between processes i and j. If the link $(i, j) \notin \Delta_l$ then $a_{ij} = 0$. Also we take $a_{ii} = 0$, $\forall i$. The communication cost can be thought of as being composed of two parts: a ***static*** part that takes into account the total number of bytes transferred in a single execution of processes i and j, and a ***dynamic*** part that accounts for the frequency of process execution and communications related queueing delays. Note that whereas the static communication cost can be precisely accounted for by inspection of the processes code, the dynamic part has to be estimated based on gathered statistical information and is dependent upon the allocation itself!

Next we model the interconnection of processors through a delay matrix D where the typical element d_{kl} denotes the communication delay for sending a byte (a message of unit length) from processor k to processor l. If processors k and l are neighbors in the processor graph, then d_{kl} reflects the cost of point-to-point or multiple access communication. If the two processors are not neighbors, but there is a path from k to l in the processor graph, then d_{kl} reflects the routing cost for sending a byte from k to l, dependent of course on the particular routing algorithm used. Finally, if there is no path between k and l, then $d_{kl} = \infty$, and if $k = l$, then d_{kl} reflects the cost of *intra-processor* message sending per unit message length. If a particular allocation assigns process i on processor k and process j on processor l , then the communication cost for this particular assignment is taken to be $a_{ij}d_{kl}$.

Each process i represents a ***load*** l_{ik} on processor k which loosely reflects this process's demands on processor resources such as CPU time, memory, I/O devices (but not those dedicated for inter-processor communications) and secondary storage. Each processor k sets an upper bound U_k on the total load that can be allocated. In addition, since we want to balance the load allocated to the processors of the distributed system, we also set a lower bound L_k on the load allocated to each processor k. Note that by setting $l_{ik} = \infty$ for some of the k's , we can model a process's preference in being assigned to specific processors only. Some researchers have proposed algorithms for allocating processes to processors under these preference constraints [Haes80]. Let $X_{ik} = 1$ if process i is assigned to processor k and 0 otherwise. Then the processor capacity and load balancing constraints can be written as:

$$L_k \leq \sum_{i=1}^{i=n} l_{ik} X_{ik} \leq U_k \quad \forall k,\ k = 1, \ldots, N \qquad [2.1]$$

where n is the total number of processes and N is the total number of processors. In addition, since a process can be assigned to only one processor, we have the following constraint:

$$\sum_{k=1}^{N} X_{ik} = 1, \ \forall i, \ i = 1, \ldots n \qquad [2.2]$$

The minimization of the communication cost can now be written as:

$$\min \sum_{i}^{n} \sum_{j}^{n} \sum_{k}^{N} \sum_{l}^{N} a_{ij}\, d_{kl}\, X_{ik}\, X_{jl} \qquad [2.3]$$

For the allocation algorithm discussed in this paper, we make the following simplifying assumptions: we treat all processes as equals in terms of the load that they represent to the processors. Typically, in large transaction processing centers there is a small number - probably no more than one hundred - of very active and resource demanding transaction types (processes) and a large number - probably in the thousands - of rarely, if ever, used transaction types. Only the allocation of the active transaction types has an impact on the performance of the center, not the allocation of the remaining thousands. We assume that these active processes represent roughly equal workloads, and we normalize their load to one. We call this assumption the "unit workload assumption". Furthermore, we assume that $d_{kk} = 0\ \forall k,\ k = 1, \ldots, N$, i.e. that intra-processor message passing incurs a negligible communication cost as compared to the inter-processor one. We feel that this is a fully justified assumption for all existing distributed systems.

In the next two sections we describe our approach to process allocation. We propose the following combination of heuristic techniques: making use of a clustering algorithm, organize the graphs Δ and Π in hierarchies of clusters using the weights on their links as the clustering (similarity) measure. Call Σ and T the resulting process and processor cluster trees respectively. In 4.0, Allocation Algorithm, we show how to map the nodes of Σ to the nodes of T . This mapping defines an assignment of processes to processors. The basic idea of the clustering algorithm is to start with a weighted graph where each node represents a lowest-level cluster (a leaf in the associated cluster tree). We then form clusters of the next higher level by grouping pairs of nodes connected with links of maximum weight. These pairs are then considered single nodes for the next iteration of the algorithm. Termination occurs when there is only one node left, the root of the cluster tree.

3.0 Clustering

The algorithm presented is a variation of the agglomerative algorithm [Hago83] which uses the weight between nodes as the similarity criterion. The reader can consult [Bowe87] for a formal definition of the algorithm. The input to this algorithm is a graph $G = (V,A)$ where V is a set of nodes and A is a set of edges. Associated with A is the set E, which contains the weights of all edges in A.

The first step is to copy the graph G into a working graph G' . Since an agglomerative algorithm is used, a series of intermediate passes are made which cluster the nodes with the greatest affinity. During each intermediate pass, each node must be put into a cluster. A cluster is a collection of nodes, possibly a single node in the case of a disjoint node (i.e., not communicating with any neighbors). Once an intermediate cluster has been formed, all nodes but one are removed from the working graph (G'). The single remaining node acts as a representative for the removed nodes during subsequent iterations of the algorithm. An intermediate pass terminates when all nodes have been clustered, and the algorithm terminates when G' contains a single node.

The first step of the intermediate pass is to select a *pivot* node. This node is the one adjacent to the largest edge in the graph. Since there are at least two of these nodes, we can break ties by greatest number of edges or by selecting the lowest numbered node. The next step is to select all the neighbors of the pivot node which have not yet been clustered. These are sorted into descending order based on the edge weight between the neighbor and the pivot. The set of all neighbors and the pivot node are now considered for clustering into a single node. A threshold value is used to select neighbors (and neighbors of neighbors up to depth k) with approximately equal weights e_{ij}.

An array c is introduced to track when nodes have been clustered. Before an intermediate pass is begun, $c(v_i)$ is set to zero for all nodes in V'. Once the pivot and neighbors have been selected, the pivot node is marked as clustered by the $c(v_i)$ variable being set to one. However, the other elements of the cluster do not have to have their $c(v_i)$ adjusted since they are entirely removed from the set V'. The graph must also be changed to reflect the clustering. All of the neighbors selected for the cluster are removed from V' and all links from the pivot to all other links *within* the cluster are removed from A'. Since the pivot must represent all the deleted nodes for the remainder of the algorithm there are several adjustments to the edge values which must be made (see details in [Bowe87]).

4.0 Allocation Algorithm

The allocation requires a process tree for input. Non-leaf nodes in this tree are denoted by r, and for any node r there is an associated value, $n(r)$, which is the number of children of r. With each node r, we associate a set of values w_i which is the number of processes represented by child i of r. An example of a process cluster tree is shown in Figure 1. The leaves represent the actual processes and the interior nodes are the result of the clustering and represent the clusters of processes. Each interior node has i children and each of these children have an associated w_i value which is equal to the number of leaf nodes under the particular child.

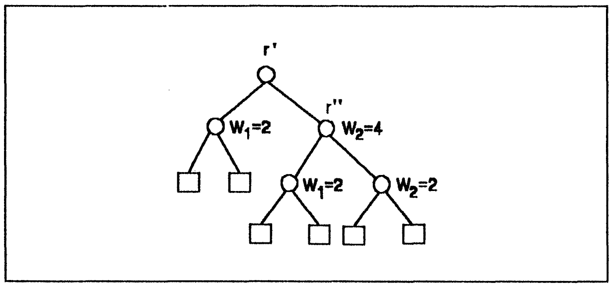


Figure 1. Example of a Process Cluster Tree

The allocation algorithm also requires a processor tree for input. Non-leaf nodes in this tree are denoted by R and for any node R, there are associated values m_i and M_i. These values represent the aggregate minimum and maximum workload balancing constraints, respectively, of child i of node R. An example of a processor cluster tree is shown in Figure 2. Associated with each leaf (a processor) is an L_k and a U_k which represent the minimum and maximum workloads, respectively, allowed for the processor k. An interior node k has m_k and M_k defined as follows:

$m_k = \sum L_k$, where k is a leaf of the subtree rooted at R
$M_k = \sum U_k$, where k is a leaf of the subtree rooted at R

We define the violation, V_i, of a node i of the processor cluster tree as follows:

$$V_i = \frac{m_i - c_i}{m_i}\text{ , } c_i \text{ is the current assigned workload on } i$$

The V_i's are then partitioned into the sets $S_v = \{V_i | V_i > 0\}$ and $S_a = \{V_i | V_i \leq 0\}$. Note the following relationships between the current and minimum workloads:

$$0 \leq c_i < m_i \quad \Rightarrow \quad 1 \geq V_i > 0\text{, thus } V_i \in S_v$$

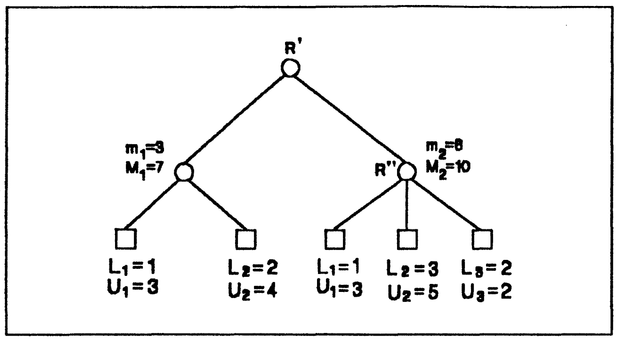


Figure 2. Example of Processor Cluster Tree

$$m_i \leq c_i < M_i \quad \Rightarrow \quad 0 \geq V_i > \frac{m_i - M_i}{m_i}\text{, thus } V_i \in S_a$$

This means that the set S_v contains all the processors which have not yet met their minimum workload requirement, and are thus in violation of the constraints. The greater this value (with a maximum of one) the ***farther*** they are from meeting their minimum requirement. The set S_v is sorted into descending order so that the first element of S_v is always the processor which is assigned work first. Once a processor has been assigned work more than the minimum, then the corresponding V_i becomes negative, V_i is removed from S_v and added to the set S_a.

The set S_a is referred to as the auxiliary set, and contains processors which have met the minimum workload requirement but have not met the maximum allowed workload. These processors are still available for assignment. S_a is sorted into ascending order based on the absolute value of V_i, which means that the first element is the one which is ***farthest*** from the maximum allowed workload.

We next present the allocation algorithm. It assumes both the process graph and processor graph have been clustered into trees. The details of the algorithm are shown in Figure 3. The allocation algorithm is called with the parameters (r,R), where r is a node in the process tree and R is a node in the processor tree. For the initial call, (r,R) are the roots of these respective trees. To allocate processes to processors, the algorithm maps the process tree onto the processor tree. The process tree is altered so that r is made to have the same number of children as R, that is, child i of r (process) is assigned to child i of R (processor).

The first step in the algorithm (only when both r and R are roots) is to verify the feasibility of the solution. If the total number of processes does not lie between the total minimum and maximum workload constraints (summed over all processors) then no solution exists. The second step of the algorithm is to check if R is a leaf in the processor tree. Mapping the children of r to the children of R creates the assignment. If R is a leaf, and thus a single processor, then all the children of r have been assigned to processor R. Since the algorithm is implemented recursively, this check simply terminates a path in the algorithm.

```
ALLOCATE(r,R)

1.  If $r$ and $R$ are roots, then check global constraints.
    If these fail then stop; there is no solution to the problem.
        $\sum_{i \in children(r)} m_i \leq n \leq \sum_{i \in children(R)} M_i$
2.  If R is a leaf then RETURN
3.  $S_v = \{V_i \mid V_i = 1, i \in children(R)\}$
4.  $c_i = 0, \forall\ i \in children(R)$
5.  $P_r = \{w_i \mid i \in children(r)\}$
6.  $RA_i = \{\}$ for $i \in children(R)$
7.  Repeat until $P_r = \{\}$ or $S_v = \{\}$
    a.  Let $i$ be child of R with maximum value of $V_i$
    b.  SELECT child $j$ of $r$ to be assigned to processor $i$
    c.  Perform updates
            $c_i = c_i + w_j$
            $V_i = \frac{m_i - c_i}{m_i}$
            If $V_i \leq 0$ then remove $V_i$ from $S_V$ and if $c_i < M_i$ then add
            $V_i$ to $S_a$
    d.  Remove $w_j$ from $P_r$
    e.  Insert $j$ into $RA_i$
8.  If $P_r \neq \{\}$ then REPEAT UNTIL $P_r = \{\}$
    a.  Let $i$ be child of $R$ with MIN $|V_i|$ in $S_a$
    b.  SELECT child $j$ of $r$ to be assigned to processor $i$
    c.  Perform updates
            $c_i = c_i + w_j$
            $V_i = \frac{m_i - c_i}{m_i}$
            If $c_i = M_i$ then remove $V_i$ from $S_a$
    d.  Remove $w_j$ from $P_r$
    e.  Insert $j$ into $RA_i$
9.  Now $P_r = \{\}$ since all elements of $P_r$ have been assigned to $RA_i$.
    For each set $RA_i$ create a new node $i$ as a child of $r$ and make
    all nodes in $RA_i$ children of $i$.
10. Now $n(r) = n(R)$. ALLOCATE(i,i) for $i = 1, 2, \ldots n(R)$
```

Figure 3. Allocation Algorithm

Before the main loop of the algorithm there are three initialization steps. In the first, each V_i in the set S_v is initialized to one. A $V_i = 1$ simply implies that the current assigned workload (c_i) for that processor is 0. Next, the set P_r is initialized to contain the work units represented by each child of r. Finally, for each child i of R there exists a set (RA_i) which contains the indices of the processes (e.g. j for child j of r) which have been assigned to this particular processor (or cluster of processors).

The main loop of the allocation algorithm begins in step 7. The *repeat until* clause causes the algorithm to repeat until all of the processes (children of r) have been assigned ($P_r = \{\}$) or all of the processors have at least met their minimum workload requirement (thus $S_v = \{\}$). First select the processor with the greatest V_i value (i.e., we select the processor which is *farthest* from meeting its minimum workload constraint). Once a processor has been chosen, the routine *SELECT* is called to pick one of the remaining children of r to be assigned to this processor. This routine is described in the following section. After the selection has been made, the rest is just bookkeeping. The current workload of the selected processor (c_i) is updated to reflect the work assigned (w_j), and a new V_i value must be calculated for the processor. If V_i is negative, then the processor has met its minimum workload constraint, and that V_i is removed from S_v. In addition, if $c_i < M_i$, that V_i is added to S_a. Finally, the index (j) of process child j is added to the set RA_i.

If S_v becomes empty before P_r, then all processors have met their minimum workload requirement but there are still processes which have not yet been assigned to processors. Step 8 is conceptually the same as Step 7, but the target set of processors is different. Since $S_v = \{\}$, the set S_a is used to select a processor for assignment. From this set, the processor with the smallest $|V_i|$ is chosen (i.e., the processor which is *closest* to the minimum workload). The remainder of this step is the same as step 7, except that S_a is updated instead of S_v. Elements are removed from S_a when they have been assigned their maximum workload.

Step 9 modifies the tree structure for the children of r (processes). A new set of children are created under r equal in number to the children of R. For each new child i of r, the nodes assigned to processor-i (elements of RA_i) are made to be children of this new child i of r. If only one child of r is assigned to a particular processor then this intermediate node can be discarded.

The final step is to recursively call the algorithm. At this point, the children of r have been assigned to children of R so that both r and R have the same number of children. For each of these assignments, a call to *ALLOCATE(i,i)* is made for each of the pairs of children.

The unit workload assumption and the check for the minimum workload constraint during the SELECT process guarantee that the algorithm terminates with a solution. If the unit workload assumption were to be dropped, then modifications would have to be made to ensure that the workload constraints are satisfied.

4.1.1 SELECT

This routine selects child j of r (cluster of processes) with the maximum w_j to be assigned to processor i. *SELECT* checks whether there are still enough processes in P_r to meet the minimum workload requirement of all processor nodes that are children of R, after the removal of j from P_r.

By way of example, assume that the processes in Figure 4 are to be assigned to two clusters of processors, each with (75,85) as their workload constraints. Figure 5 shows the allocation after process children 1 and 2 have been assigned to processor cluster nodes 1 and 2, respectively. Note that RA_i and c_i reflect these assignments.

If the allocation algorithm did not examine the implications of these assignments, there could be cases where the algorithm would not terminate with a solution. In this example, process child 3 of r ($w_3 = 40$) would be assigned to processor child 2 of R ($V_2 = 0.4$). These nodes are selected because they have the maximum violation (processor child 2) and workload (process child 3). This

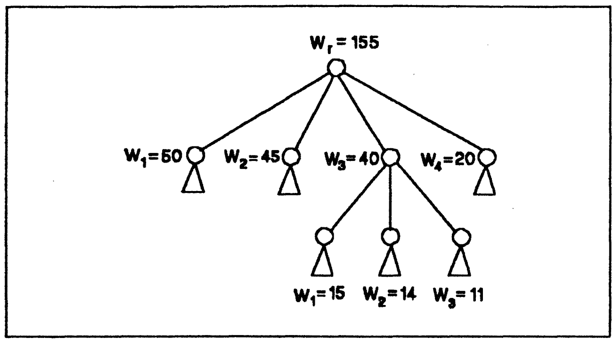


Figure 4. Process tree before a pop.

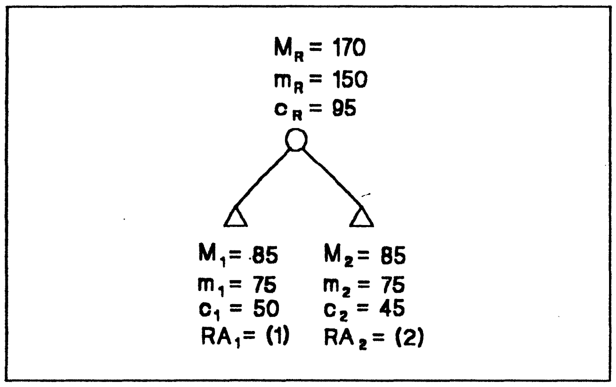


Figure 5. Allocation after first two assignments

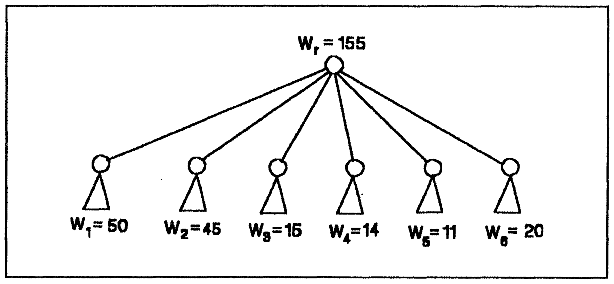


Figure 6. Process tree after a pop

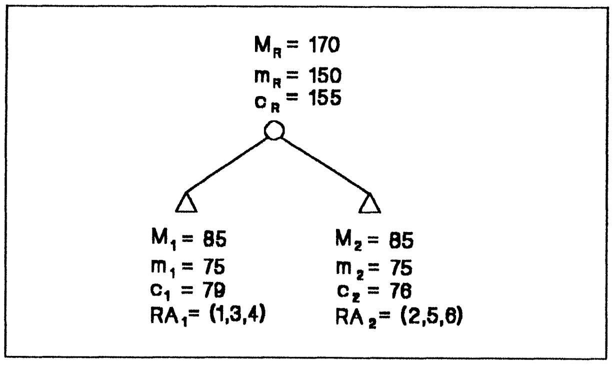


Figure 7. Final allocation

would make $c_2 = 85$ and then force process child 4 to be assigned to processor child 1 giving a final $c_1 = 70$. The assignment would have failed since the minimum workload constraints of processor 1 is not met. Therefore, the algorithm selects the greatest w_j value, updates P_r and the c_i value for the selected processor, and makes the following check:

$$\sum_{w_m \in P_r} w_m \geq \sum (m_i - c_i) \;\; \forall i, c_i < m_i, i \in children(R)$$

The left hand side of this expression calculates the total amount of *unassigned* work (assuming that w_j has been removed from P_r). The right hand side calculates the total amount of work required to allow all processors to meet their minimum workload constraint. If this check is not satisfied, then the process node with the greatest w_j value is *popped* and the assignment is attempted again after selecting a new w_j. If child 3 of r is popped, then the resulting structure is shown in Figure 6. Notice that the original node (with a $w_3 = 40$) has been removed and the three children of the former child 3 are now children of r. This completes a pop.

Continuing with the example, after child 3 of r is popped, the new children of r are assigned to the processor tree as shown in Figure 7.

5.0 Results

The generation of a realistic workload is critical for the validation of the allocation algorithm. However, little is known in the literature about "typical" process graphs, with the possible exception of parallel algorithms solving specific numerical problems modelling highly localized interactions between neighbors (e.g. FFT, finite element methods, and so on). Very little information is available about process communication patterns encountered in distributed systems consisting of collections of workstations, departmental computers and possibly mainframes. Our technique is to first build a library of small subgraphs (2 - 8 process nodes), representing highly interacting processes, and then randomly group the subgraphs, through low volume communication links, to make the larger process graph. We have chosen subgraphs that, we believe, accurately reflect patterns of heavy communication among groups of processes in existing distributed systems. These subgraphs are described next.

Pipeline. The first set of graphs consist of the *pipeline* graphs which represent a set a processes with communication in a linear fashion with at most two neighbors. Pipes are frequently used by processes running on UNIX and MS-DOS, for instance. Our library contains seven different pipeline lengths: 2, 3, 4, 5, 6, 7 and 8 processes long. The weight on a pipeline link can take the value 5, 10 or 100. For any particular library entry, all link weights are the same. We therefore have 7 x 3 = 21 different pipelines in our library.

Ring. The *ring* is a slight modification of the pipeline graph such that the last node communicates back with the first node as shown in Figure 9. There is a total of 18 different rings in our library having 3, 4, 5, 6, 7 or 8 nodes and 5, 10 or 100 as weights.

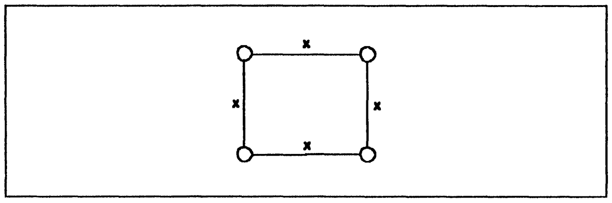


Figure 9. A Ring Subgraph with Four Nodes

Server. The *server* subgraph represents a server process which communicates with several processes that submit requests, as shown in Figure 10. Examples of server processes are file servers, name servers and so on. Although there may be a large number of processes occasionally communicating with a server in a distributed system, this subgraph models the situation of some processes making heavy use of a particular server. It may then make sense to allocate these heavy users on the processor where the server resides. There are 15 different server subgraphs in the library (4, 5, 6, 7 or 8 nodes and 5, 10 or 100 as weight values).

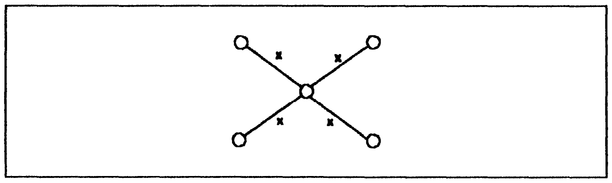


Figure 10. Server Subgraph

Interference. The *interference* subgraph represents a contention situation. The weight of the edges do not now represent volume of data transferred but serve instead as a measure of, for example, lock contention. A weight could for example be defined as the ratio of contended locks versus the total number of lock requests over a time unit and for a particular pair of processes. A data sharing model between processors is assumed here. The subgraph implies that it is highly desirable for all the processes to run on a single processor. Any process that is scheduled on a remote processor interferes with *all* other processes of the subgraph, as shown in Figure 11. There are 15 interference subgraphs in the library (4, 5, 6, 7 or 8 nodes and 5, 10 or 100 as weight values).

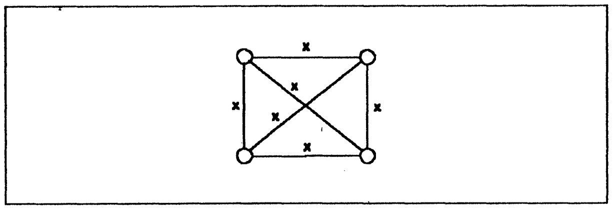


Figure 11. Interference Subgraph

Full process graph generation. The full graphs were constructed from the 21 + 18 + 15 + 15 = 69 subgraphs. We experimented with process graphs consisting of 45,90,180 and 360 nodes, respectively. For a particular number of nodes (e.g. 90 nodes) 100 full graphs were generated. The 400 process graphs (100 of each set of 45, 90, 180 and 360) are constructed by randomly selecting subgraphs until the desired number of nodes have been generated. Each subgraph is connected through a *weak link* of weight one to the subgraph previously selected, and the last subgraph is connected back to the first subgraph.

Since the subgraphs are connected into a circular fashion, it is possible to estimate a lower bound on the optimal solution. Assume a number of processors less than or equal to the number of subgraphs in the process graph, fully interconnected through a homogeneous interconnection medium (say a bus or a crosspoint switch), and 2 and ∞ as low and high thresholds of utilization. Then the optimal assignment would be to allocate at least one subgraph on each processor and the communication cost would be N, the number of processors. If the number of processors is increased to be greater than the number of subgraphs in the process graph, or if the high threshold is decreased, then it is possible that a subgraph may have to be allocated on two or more processors in the optimal solution and the communication cost will be higher than N. But the communication cost never drops below N, if we do not allow idling processors (having 0 as low threshold of utilization). This observation proved to be very helpful in evaluating the heuristic solutions.

The allocation problem, formulated as a 0-1 linear integer programming problem, was solved for a number of different inputs using MIPS/370. Figure 12 compares the optimal solution to our heuristic algorithm. The cost shown is that of the network communication. The worst case cost is an assignment where all communication costs are incurred. Clearly, our heuristic algorithm produces very reasonable allocations very fast.

	2 processors 90 processes (L,U) = (40,45)	4 processors 90 processes (L,U) = (20,24)
Heuristic Time	0.581 sec	0.592 sec
Optimal Time	4.52 sec	1777.22 sec
Heuristic Cost	3.47	39.26
Optimal Cost	2.00	4.20
Worst Case Cost	4245	4820

Figure 12. Heuristic and optimal allocation, CPU time and communication cost

Next we present the results from 3,200 runs of the clustering and allocation algorithms. There are four sets of process graphs which consist of 45,90,180 and 360 process nodes. Each set contains 100 graphs (i.e., there are a total of 400 process graphs). Four fully connected processor graphs with all weights equal to one were used in the experiment (2,4,8 and 16 processors). For each matching of a set of process graphs to an individual processor graph, the clustering and allocation algorithms were executed with four different workload balancing constraints. We define the *target* workload to be that of a *uniform* workload distribution equal to the number of processes divided by the number of processors. We further define a set of variances (10%, 25%, 50% and 98%) which are applied to the *target* workload to obtain minimum and maximum workload bounds.

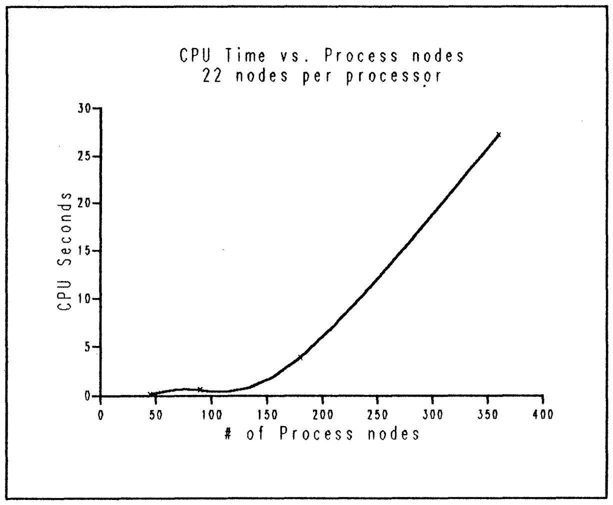


Figure 13. CPU Analysis

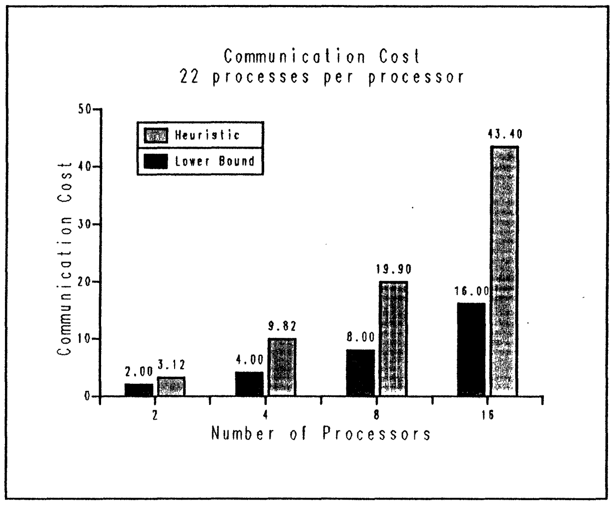


Figure 14. Communication cost for 22 processes per processor

The CPU consumption is a critical measure of this algorithm. In Figure 13 we have graphed the CPU seconds, on an IBM 3090-200, required for the clustering and allocation algorithm for a variety of process nodes. The aim of the configurations used in this example was to show the growth in CPU consumption as the size of the system grows. For this reason we have decided to use processor configurations which always get approximately 22 process nodes per processor (i.e., 90 processes on 4 processors, 360 processes on 16 processors). The best fit for the measured CPU time is the following function:

$$\textbf{CPU Seconds} = k \times n^{2.767}$$

where $k = 2.031 \times 10^{-5}$ and n is the number of process nodes. This relationship shows that the algorithms are practical even for large configurations.

Again looking at processor configurations which always get approximately 22 process nodes per processor (i.e., 90 processes on 4 processors, 360 processes on 16 processors), we have also studied the communication cost. Since the optimal solution is extremely time consuming to calculate, we have used a feature inherent to the process graphs to compare our solutions. The workload graphs were generated by connecting a collection of small (2-8 process nodes) nodes together to form a large process graph. These small graphs were connected by a "weak-link" of value one. One can intuitively imagine the optimal solution to assign these small subgraphs to a single processor. Then the only interprocessor communication is due to these "weak-links". Therefore we can state that a lower bound for the optimal solution is the number of processors in the configuration. Figure 14 shows the heuristic solutions for the same points that were shown in Figure 13. These numbers are the average of 100 runs on each given processor configuration. In each of the 100 runs a different process graph is being used for input. These results are extremely promising since they are very close to the lower bound of the optimal solution.

In addition to looking at a fixed number of processes per processor, we also present results for taking the same set of processes and assigning them to a varying number of processors. In Figure 15, 90 processes are assigned to 2,4, 8, and 16 processors. The workload bounds used for these points were 50%. Note that the assigned number of processes per processor gets smaller as the number of processors increase. These results again show that the algorithms are making very good assignments.

As stated in the introduction of this section, the workload bounds were varied to determine the sensitivity of the algorithms to these parameters. The CPU time was not at all sensitive to the workload bounds but we found the solutions themselves to be rather sensitive. Figure 16 shows communication costs for the case where 180 processes are assigned to 8 processor nodes. The workload bounds were calculated using variances of 10, 25, 50 and 98 percent. The results are not immediately intuitive. Using a variance of 10% the workload bounds are very tight (20-24 processes per processor). This case produces the worst results because the allocation algorithm is forced to split apart most of the subgraphs in the workload process graph. Variances of 25 and 50 percent produce good results because the allocation is able to maintain the affinity of the subgraphs.

One would think that using a variance of 98% would produce results better than using 25 and 50. This is not the case. As shown in the graph, this case produces worse results. Consider the following example. Assume that the process tree root had four subtrees each representing 44 units of work. Also assume that the processor tree root contains eight subtrees each with workload bounds of (1,44). The allocation algorithm would assign the first three subtrees from the process graph to the first three subtrees of the processor graph. This would leave one process subtree of 44 nodes to be assigned to five processors. The single process subtree would now have to be "popped" several times to meet this allocation. Having tighter bounds, primarily the *upper* bound, produces more popping at the higher levels of the processor tree. The clustering has produced a tree with heavily communicating nodes at the bottom of the tree, therefore, popping at the top of the tree does not create a severe communication cost penalty. Once the popping is forced to go deep to the bottom of a process tree, the communication penalties become high.

The workload generations for these experiments produce a series of subgraphs (2-8 nodes) which are connected by a "weak-link" of weight one. It is interesting to measure the communication cost when the workload bounds are made so tight that these subgraphs must be split onto separate processors. In Figure 17, the 90 node process graphs are assigned to 16 processors with all four workload variances. The target assignment for each processor is 5.6 processes per processor.

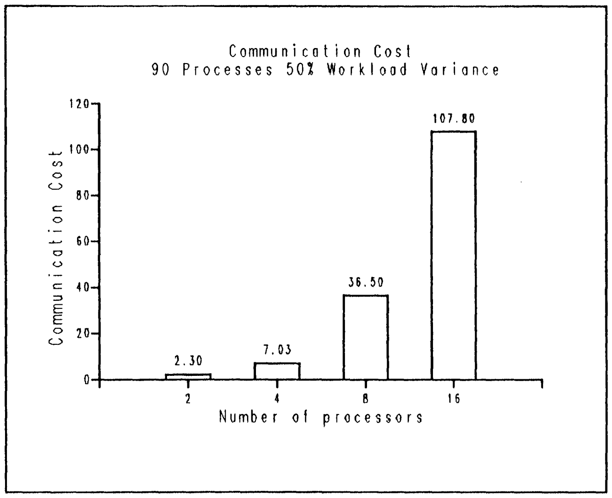


Figure 15. Communication cost for 90 processes assigned to 2,4,8 and 16 processors

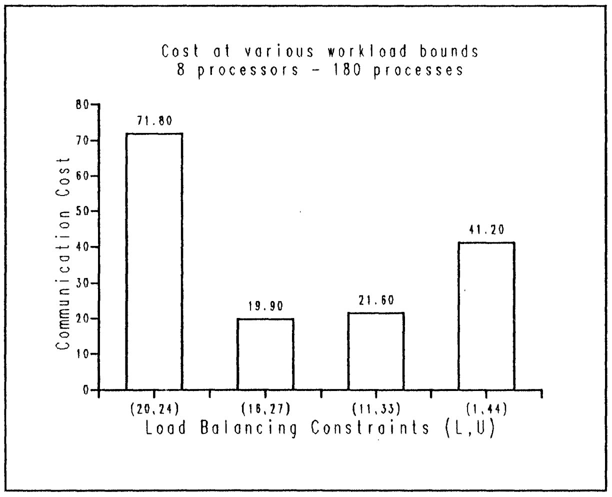


Figure 16. Sensitivity to workload bounds

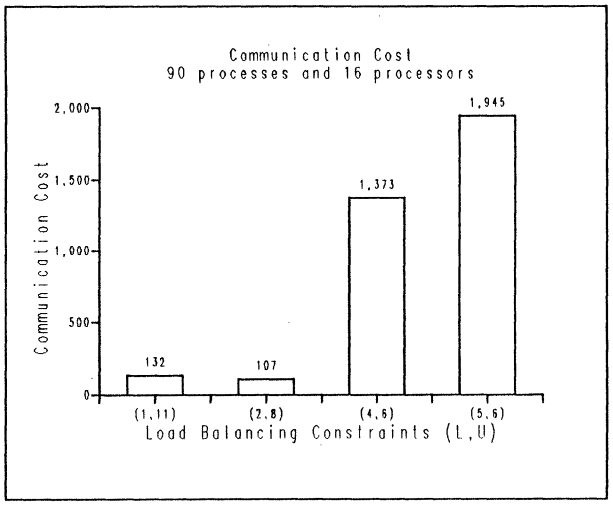


Figure 17. Effects of tight workload bounds

In the first two cases (10% and 25% variance) the communication costs are very reasonable. This is because the subgraphs are still able to be mostly assigned to one processor. Once the maximum workload bound goes below the subgraph size the solutions get dramatically worse. It should be noted, however, that even in the worst case (bounds of 5,6) the communication cost of 1,945 is still reasonable. The total amount of communication for these processes nodes is 4,569. Although we could not calculate the optimal solution for this last case, we suspect that the heuristic solution is considerably closer to the optimal than to the worst case.

6.0 Conclusion

In this paper we presented a novel approach that clusters groups of processes that communicate frequently and allocates them as a group to processors of a distributed system. We also described a methodology to build a library of workloads that were used to evaluate our clustering and allocation algorithms. This methodology allows us to experiment with a great variety of different components, modelling groups of processes exhibiting specific patterns of communication, by combining them randomly to form realistic large process or processor networks. We believe that this workload accurately represents component processes found in distributed operating systems.

Using our process and processor graph generating methodology, we executed both our heuristic clustering and allocation algorithms, and a mathematical programming algorithm to yield optimal solutions. The comparison between the two is very encouraging because, at significantly lower execution time, we obtained allocations with cost very close to the optimal. More research should be done to experiment with a bigger variety of graphs, and to extend the allocation algorithm to cover the case of non-unit workloads and strong processor affinity.

Acknowledgements

John Forrest has been very helpful in showing us how to best use MIPS/370. Leo Georgiadis edited early drafts of the paper and helped us with discussions and comments.

REFERENCES

Bowe87. Nicholas S. Bowen, Christos N. Nikolaou, and Arif Ghafoor, Workload Allocation for Distributed Systems, IBM Research, RC 13180, October 1987.

Ferg86. Ferguson, D., Leitner, G., Nikolaou, C., and Kar, G., "Relocating Processes in a Distributed Computer System," *Proc. of the 5th Symposium on Reliability in Distributed Software and Database Systems*, January 1986.

Haes80. Haessig, K. and Jenny, C.J., An Algorithm for Allocating Computational Objects in Distributed Computing Systems, IBM, Zurich Research Laboratory, RZ 1016, 1980.

Hago83. Hagouel, J., Issues in Routing for Large and Dynamic Networks, IBM Research, RC 9942, 1983.

Ligg81. Liggett, R.S., "The Quadratic Assignment Problem: an Experimental Evaluation of Solution Strategies," *Management Science*, vol. 27, no. 4, pp. 442-458, April 1981.

Niko86. Nikolaou, C., Ferguson, D., Leitner, G., and Kar, G., "Allocation and Relocation of Processes in a Distributed Computer System," in Yemini, Y., editor, *Current Advances in Distributed Computing and Communications*, vol. 1, 1986. Computer Science Press.

A DISTRIBUTED APPLICATION FOR THE PHARROS PROJECT (PHARROS = PARALLEL HETEROGENEOUS ARCHITECTURE, RELIABLE REALTIME OPERATING SYSTEM)

Tom Geigel and Mike Pagan
GE/RCA Advanced Technology Laboratories
Moorestown, New Jersey 08057

ARPANET: tgeigel%henry.decnet@GE-CRD.arpa
mpagan%henry.decnet@GE-CRD.arpa

Abstract -- RCA Advanced Technology Labs is developing an operating system kernel designed for distributed execution of programs across a network of parallel- and uni-processors. PHARROS (Parallel Heterogeneous Architecture Reliable Realtime Operating System) is being designed to be an extention of UNIX that focuses on fault tolerance and real time processing on heterogeneous processors. Using a dataflow graph (DFG) metaphor to define the application programs, DFG nodes representing processes are scheduled and dispatched via a centralized controller. This phase of the project involves linking a DEC μVAX, a Thinking Machine Corporation Connection Machine, and a BBN Butterfly computer via TCP/IP over Ethernet. This paper discusses the design and testing of the PHARROS with a sonar processing application program.

Introduction

Advanced communications and sensor equipment, such as radars, sonars, and satellite groundstations are beginning to rely on parallel processors to achieve better performance in both signal and data processing. The type of processing that must be performed, however, is very diverse; note the distinction between "signal processing" and "data processing," which has been made since the early days of the field. Traditionally, the data processor has been a conventional uniprocessor, the signal processor has been a "conventional" array processor, and both have been tied together on a single bus. More recently, the search for higher performance has lead to algorithms which must run on complex heterogeneous systems of processors. Systolic arrays, shared memory multiprocessors, special purpose Lisp machines, and even more esoteric beasts are finding their way into such systems. One of the goals of the PHARROS project is to provide communication and application software system designers with a layer of abstraction to shield them from the unnecessary details of the hardware. This, of course, should be the goal of all operating systems. The uniqueness of PHARROS is its focus on heterogeneity, fault tolerance, and realtime processing.

Objectives

PHARROS itself is intended to be an extension to the UNIX kernel. It is not a full-blown operating system, but rather it is to be an infrastructure for the dataflow style of programming under an existing operating system. In many ways it resembles NETlinks [1], Sun RPC functions, the Warp Programming Environment (WPE) [2], and other such remote program execution facilities, in that its main purpose is to allow programs to run and data to be transferred across different machines on a network.

The major objective of PHARROS is to provide the programmer with a consistent view of the system on which it is running, a system comprising a number of dissimilar processors. All of the various computers and operating systems must present the same basic services which behave the same way and produce the same results. To insure that this would be the case, a common interface was derived from the ACOS/ECOS DFG specification [3] which is summarized in Table 1. ACOS/ECOS (ASP Common Operational Support Software/EMSP Common Operational Support Software) defines all of the parameters necessary to fully specify a dataflow graph. It was originally developed to allow signal processing applications to be described in DFG form and automatically converted into executable software modules.

TABLE 1. ACOS/ECOS DFG SPECIFICATIONS

Data Flow Graph (DFG)	A set of nodes representing the processing to be performed connected by a set of arcs representing the directed information flow through the graph.
Node Parameters	Name: The name of the executable code associated with a node. Inputs/Outputs: Connections to arcs.
Arc Parameters	Inputs/Outputs: Connections to nodes. Read Amount: The amount of data the node will read from a given arc each time it executes. Offset Amount: The number of data elements to skip before starting a read on a given arc. Consume Amount: The number of data elements to be removed from an arc on each read. (NOTE; the read amount does not necessarily equal the consume amount) Produce Amount: The number of data elements produced whenever the node associated with the arc is executed. Threshold Amount: The minimum number of data elements required for the node associated with the arc to fire.
SPGN (Signal Processing Graph Notation)	A generic high-level language defined specifically to describe the structure of a DFG.

ACOS/ECOS DFG's consist of a set of nodes representing software modules connected by arcs which represent unidirectional communication paths. Nodes are best thought of as black-box subroutines with no connections to the outside world other than their input/output arcs, and no retained state information between executions. Arcs are thought of simply as queues, regardless of how they might be implemented. This abstraction should allow the development of individual nodes "in a vacuum," that is, without regard to the state of development of any other nodes in the graph. The strict enforcement of node input/output constraints imposed by this specification makes this possible by forcing the programs to be written in a very modular fashion.

ACOS/ECOS also provides a definition of a high level language for describing the structure and function of a graph, but PHARROS uses only the overall DFG description sections of the specification which define the node firing and arc production/consumption parameters.

The first phase of this project, completed in December 1987, was to demonstrate the feasibility of PHARROS on an existing network of different computers running different operating systems. At the end of this phase, half of the PHARROS acronym was fulfilled: it is both "Parallel" and "Heterogeneous." During this four month effort, an already existing signal processing algorithm was chosen and recoded to conform to the PHARROS specification. This provided both continuous feedback for debugging PHARROS and a complete demonstration of its capabilities using a known application.

The next phase is now underway, which involves converting the entire system to MACH [4] and upgrading the network hardware. At the same time the kernel will be extended further to implement the second half of the acronym: dynamic node assignment will make it "Reliable" while fast and smart scheduling will allow realtime programming.

Hardware

The machines used in the first phase of the project include 2 DEC MicroVAXen, a DEC VAX 8350, a 16 processor BBN Butterfly, and a 16K processor Connection Machine-1 from Thinking Machines Inc. The VAXen run Ultrix, while the Butterfly runs Chrysalis (a Butterfly-specific OS), and the Connection Machine acts as an extension of the VAX 8350. All of the machines are networked together over Ethernet.

Approach

This section describes the approach taken during the first phase of the PHARROS project.

Omnipotent Controllers

The intention of the PHARROS project is to create an environment for coordinating the execution of an application across heterogeneous parallel processors. This task was performed with the use of executive programs called Omnipotent Controllers (OCs) operating independently on each participating machine. The OCs, written in C, are in charge of deciding what task is to execute and on what machine, what data files must be accessed by the task, and creating and sending the message that will execute the task. Messages between OCs control the passing of files from machine to machine and the spawning of tasks from the application DFG. The OCs implement a message queue to read messages and perform operations on a first come-first serve basis. Communications between the different machine's OCs are performed via Ethernet interface programs, IPC, NL, and READIT, also written in C (see Figure 1).

Initially, the PHARROS project implements a centralized scheduling scheme that schedules application DFG nodes that have been assigned, a priori, to a particular machine. Centralized scheduling comes in the form of a Master Omnipotent Controller (MOC). The MOC is in charge of scheduling the application nodes, managing the data arcs produced by the application, and coordinating the transfer of these arcs to the correct machines. The other OCs are slaves to the MOC, performing exactly as the MOC instructs them, and notifying the MOC as to the completion of any application node running on its machine.

Messages. PHARROS operations are performed cooperatively between OCs; commands are sent via a message passing mechanism. A message consists of four parts: a message type, the message's ultimate destination, a file associated with the message, and the message size. The type of message determines how the file is interpreted by the destination OC. The list of defined messages are:

START	- start DFG
EXECUTE	- execute a DFG node
NOTIFY	- notification of a node's completion
TRANSFER_x	- transfer a file to machine x. If x is the receiving machine, then this message notifies the OC that a file has been transferred.
CREATE	- create a file
DELETE	- delete a file

When an OC wishes to send a message, it forks an IPC process, passing the message parts to IPC as arguments. IPC is a TCP/IP protocol, Ethernet communications program written in the same vein as telnet and ftp. The message is formatted and sent via a predetermined Ethernet port to NL, the IPC daemon listening on the other machine. When NL receives a message on the port, it spawns a READIT process to finish reading the transmission. When the message is read, the file part of the message is written to a file on the destination machine. The message's type and filename are then put on the OC's message queue to be read eventually by the OC.

Implementation. Because of scheduling constraints, a simple approach to implementing the application DFG was adopted. Conceptually, nodes represent executable programs that generate data. This data is stored in arcs in FIFO queue form. Since nodes can execute on many heterogeneous machines, managing arcs as a data queue becomes quite difficult. In the PHARROS implementation, arcs are treated as data files. Nodes generate data in file form; these files are listed in a FIFO queue in the Master Omnipotent Controller. In this way, the writers did not concern themselves with the issues of memory management, which was not within PHARROS's scope of interest at this time.

With data arcs implemented as files, a mechanism for binding the correct data from an arc to any particular node instantiation was easily developed. Nodes are spawned by the omnipotent controllers after receiving an EXECUTE message. An EXECUTE message's associated file contains a command line which is passed to the execve system subroutine after a fork. The form of the command line is:

```
node | in_arc_list_file | out_arc_list_file
```

The in_arc_list_file is an ASCII file containing a list of input arc files tabulated by the Master Omnipotent Controller. These filenames came from the linked lists of files representing each individual arc in the MOC's central DFG database. Likewise, the out_arc_list_file is a file of a list of output arcs for the node. Thus the node is bound to a set of data in its arcs by passing it the correct filename in the arc_list_files. From the standpoint of the node writers and DFG debuggers, this method provides a transparent and generic mechanism for reading and writing data to arcs. The node does not have to be written with arc interfacing in mind, since the binding method lends itself to producing a generic interface template for any node, given the input and output arcs. Figure 2 shows some sample code.

Master OC Scheduling. The operation of the MOC is straightforward. After receiving a NOTIFY message from a slave OC referring to a particular node, the MOC updates the node's associated input and output arcs' sizes in its database. If any arc has been consumed, the corresponding file is deleted. This is done by the MOC by sending a DELETE message to the arc file's resident machine(s). The MOC then checks the nodes associated with the completed node's output arcs, called output nodes (the output arcs are input arcs of the completed node's output nodes). If all of the output node's input arcs have enough data in them, then the node is ready to execute. The MOC creates filenames for the output data, and sends messages to the machine that the node is assigned to. These messages tell the machine to create the output arc files and execute the node. Messages are also sent to other machines to transfer the node's input arc files to the machine that will execute the node.

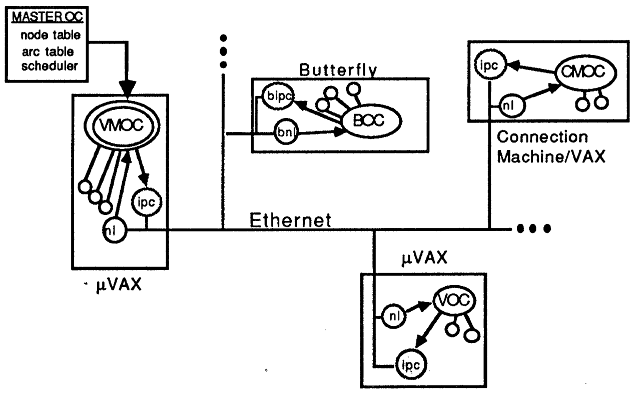


Figure 1. PHARROS Network Block Diagram

```
main (argc,argv)
    int  argc;
    char *argv[];

{
    char              filename[80];
    float             *input_data,
                      *output_data2;
    struct cmplx {
              float      real,
                         imag;
            }         output_data1[produce_amount1];
    FILE              *inargs,
                      *outargs,
                      *input_file,
                      *output_file1,
                      *output_file2;


    /*--------------- Get arc_list files from argv ------------*/
    inargs = fopen(argv[1],"r");
    outargs = fopen(argv[2],"r");


    /*----------- Get input arc filename from inargs file -----*/
    fgets(filename,80,inargs);            /* get input arc name  */
    filename[strlen(filename)-1] = '\0';/* convert \n to NULL  */
    input_file = fopen(filename,"r");   /* open arc file       */


    /*-----------------Get data from input_arc----------------*/
    input_data = (float *)malloc(read_amount*sizeof(float));
    fread(input_data,sizeof(float),read_amount,input_file);



    /*-----------------Rest of Code here----------------------*/
                    .
                    .
                    .

    /*----- Get output arc #1 filename from outargs file-------*/
    fgets(filename,80,outargs);           /* get output arc name */
    filename[strlen(filename)-1] = '\0';/* convert \n to NULL  */
    output_file1 = fopen(filename,"w"); /* open arc file       */


    /*----------------Write data to ouput_arc #1--------------*/
    fwrite(output_data1,sizeof(struct cmplx),
                                produce_amount1,output_file1);


    /*----- Get output arc #2 filename from outargs file-------*/
    fgets(filename,80,outargs);           /* get output arc name */
    filename[strlen(filename)-1] = '\0';/* convert \n to NULL  */
    output_file2 = fopen(filename,"w"); /* open arc file       */


    /*----------------Write data to ouput_arc #2--------------*/
    fwrite(output_data2,sizeof(float),
                                produce_amount2,output_file2);

}
```

Figure 2. PHARROS Interface Template

The Application

Figure 3 shows the application DFG used to test and demonstrate PHARROS. The algorithm is used to detect signals in very low signal-to-noise ratio environments. The main reasons for choosing this algorithm were its availability and the fact that it has been used a number of times by RCA/ATL to benchmark computer systems.

This algorithm is well suited for implementation using PHARROS because it consists of a number of processing stages and subroutines which map well onto widely differing computer architectures. The number crunching part of the algorithm performs well on the shared memory architecture of the Butterfly, while the preprocessing functions are better matched to the massively parallel Connection Machine. Other functions such as displaying output which support little or no parallelization are best performed by a VAX.

Conversion to PHARROS required the modules to be decomposed, with careful attention paid to parameter passing. This implementation of the ACOS/ECOS specification allows for no state to be saved in the nodes, so globally declared variables had to be eliminated. Beyond that, very few changes were needed.

Problems Encountered

A few complications hindered the implementors in producing a quick product in the amount of time given. One involved the system-imposed maximum queue size. The OCs' message queues were built using the standard UNIX message queue system subroutines. It imposed a limit of 40 messages allowed on a message queue at any one time (for the ULTRIX version). The implementers soon found out that 40 was quite inadequate, as the demonstration DFG was written to spawn 30 nodes at the same time. This resulted in greater that 60 messages being passed to the MOC at one time. Needless to say, some messages never were received by the MOC.

A quick solution involved checking the current size of the message queue prior to placing a message on the queue. This involved creating a semaphore lock so that one process at a time could read the size of the message queue at any time. This solved the problem of lost messages, but caused a greater problem that degraded the MOC's performance. If a process could not place its message on the OC's queue, it would wait a specified amount of time and then attempt it again. Soon, there were many processes attempting to read the size of the message queue, all competing with the OC for the μVAX's only processor. The result: a much slower OC which could not keep up with the ever increasing number of processes attempting to place a message on its queue the second one is removed. The solution is to increase the number or develop another message queue without implementing the UNIX routines; however, it was not a viable solution in the schedule given.

Another obvious performance degrader was the use of the file system. For every message, there was associated with it a file. This file had to be created, the filename passed to the IPC process which opened it, read it, sent it via the Ethernet to the NL daemon, which created another file to hold the file sent. Finally the OC receiving the message had to open and read the file before finally deleting it. When the message queue is redeveloped, it will contain a shared memory block capable of storing messages without the need of files.

Lastly, future work is intended to experiment with smarter, decentralized and distributed scheduling schemes and dynamic node assignment.

Results and Recommendations

The initial core of PHARROS has been demonstrated successfully after only four months from the beginning of its development. This has shown the basic concepts to be sound and reasonably easy to implement on a real network of processors. As was expected, it also showed the present network implementation to be completely inadequate for even a medium-sized demonstration application. The network is now being upgraded to improve its performance and to allow more advanced features to be added to PHARROS. In its next incarnation, all of the machines in the system will be tied into a single high speed parallel bus (such as a VME bus). Ultimately, the goal is to use the Distributed Interconnection Switch Network (DISN) to connect up the system. The DISN is a multistage switched network currently being developed at RCA Advanced Technology Labs. In addition, as was mentioned earlier, UNIX is to be replaced by Mach. Mach provides distributed processing extensions to UNIX which will make PHARROS simpler and more efficient, such as transparent interprocessor communication, multiple process and I/O threads.

Note that nodes were truly generated "in a vacuum." Previously existing, unrelated building blocks were simply spliced onto PHARROS kernel calls to produce the finished demonstration application. Debugging was accomplished by generating artificial input data and using PHARROS to present it to the node under development, which allowed symbolic debuggers to be used which might be unwieldy or impossible to link into the entire application. As soon as a node was completed, PHARROS would be used to capture its output from a dummy output arc. The output would then be used to test other nodes with real, rather than synthetic, data.

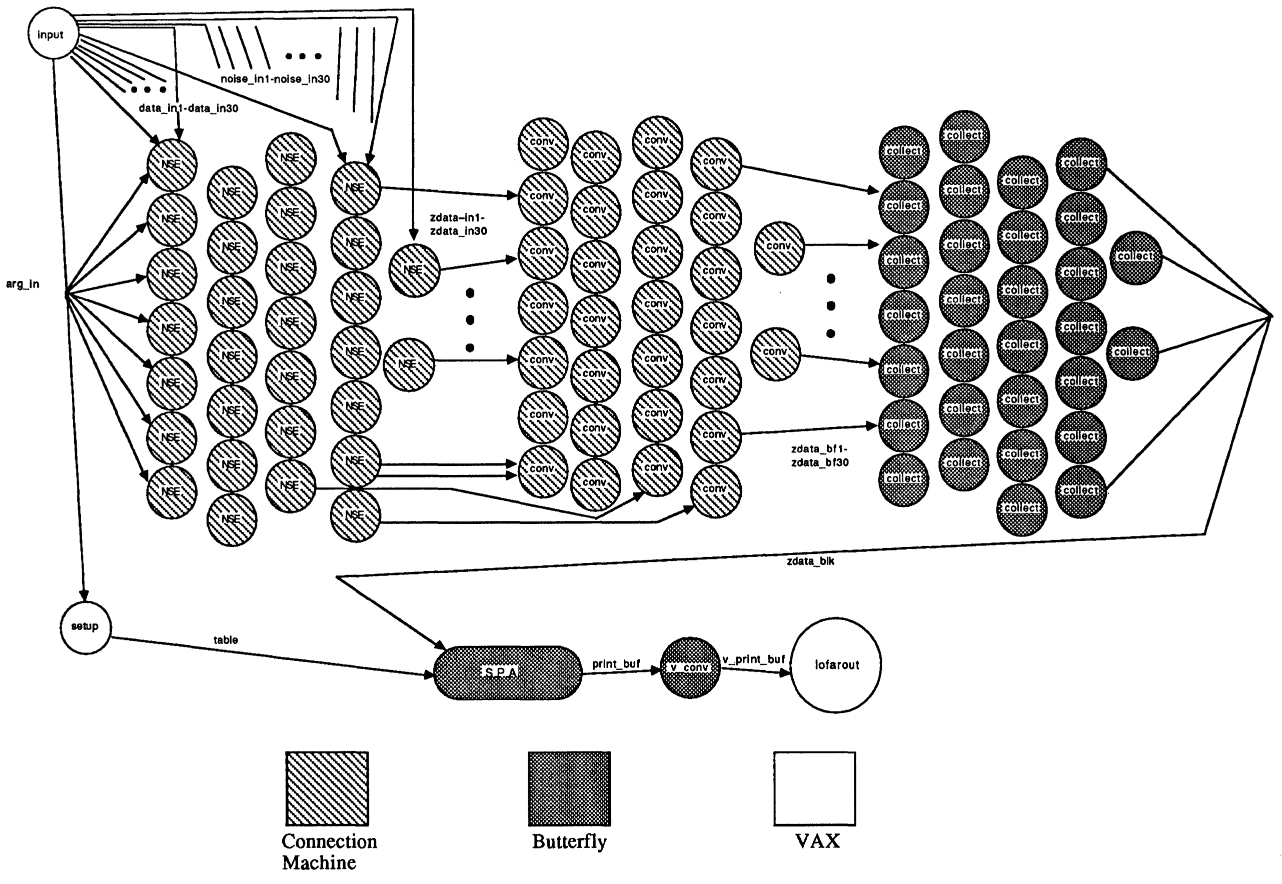


Figure 3. Demonstration Data Flow Graph

References

[1] D.F. Vrsalovic, Z.Z. Segall, D.P. Sieworek, "Netlinks: Interprocess(or) Communication Infrastructure for Distributed Environments," Carnegie-Mellon University Technical Report, Pittsburgh, PA.

[2] Bernd Bruegge, "Warp Programming Environment Users Manual," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, (September 1987).

[3] "EMSP/ASP Common Operational Support Software Methodlogy Specification," Analytic Disciplines, Inc., (June 1983).

[4] A. Tevanian and R.F. Rashid, "Mach: A Basis for Future UNIX Development," Department of Computer Science, Carnegie-Mellon University, Pitsburgh, PA, (June 1987).

[5] J. VanZandt, "Dynamic Tasking in a Heterogenerous Parallel Processor," Proceedings of MAPCON IV Conference, (February 1988).

[6] J. VanZandt, "The PHARROS Project," Proceedings of the 2nd Workshop on Large-Grained Parallelism, Carnegie-Mellon University Software Engineering Institute, Special Report CMU/SEI-87-SR-5, Ed. by Mario Barbacci, Pittsburgh, PA, (November 1987), 86 pp.

Finding Large-Grain Parallelism in Loops with Serial Control Dependencies

Henry G. Dietz

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

January 1988

This paper presents an automatic parallelization technique, **control precomputation**, whereby arbitrary sequential loops, including those with serializing control dependencies (i.e., general `while` loops) and/or irreducible flow graphs, may be transformed to permit relatively large-grain asynchronous parallel execution. The basic concept is for the compiler to isolate inherently sequential operations from the remainder of the loop, thereby enabling the remaining operations to be parallelized. Since there is a cost associated with splitting loops, not all loops can be profitably parallelized in this way. Compared to "pipelining," however, this technique effectively reduces synchronization overhead by grouping high-frequency synchronization points within a single process; the reduction in synchronization overhead is approximately proportional to the parallelism width of the machine. Such a transformation is particularly useful for non-shared memory machines, such as the current generation of hypercube computers, and with loops typical of programs written in languages like C.

1. A Brief Example

Perhaps the best way to prove that selective serialization can result in dramatic increases in usable parallelism is to give a concrete example. **Control precomputation** is a new kind of loop transformation which attempts to parallelize a portion of a loop body by selectively, deliberately, serializing some operations within the loop. In particular, operations which are involved in computing the expressions which determine when the loop is exited are placed in a separate loop called the **preloop**. The other operations are placed in a parallelizable loop called the **postloop**, and the preloop followed by the postloop are both placed inside the **closure loop**. The loop:

```
while ((c = getchar()) != EOF) {
     checksum += f(c);
}
```

Listing 1: An "Inherently Sequential" C Loop

where `f(c)` performs a relatively expensive computation with no significant side-effects, becomes something like:

```
int exit, i;

/* Closure loop */
exit = FALSE;
do {
  int ctemp[MAXWIDTH], pari;

  /* Preloop */
  i = 0;
  do {
    if (!((c = getchar()) != EOF)) {
      exit = TRUE;
      break;
    }
    ctemp[i] = c;
  } while (++i < MAXWIDTH);

  /* Postloop */
  for (pari=0; pari<i; ++pari) {
    checksum += f( ctemp[pari] );
  }

} while (!exit);
```

Listing 2: Listing 1 by Control Precomputation

which is far better than pipelining execution of the loop, because the amount of synchronization needed for execution on most machines is reduced by a factor of MAXWIDTH. It has this effect because it places the synchronization-intensive portion of the loop in a single process/processor. In this particular example, it is also unclear whether it is physically possible for the reads from a single file to be spread across multiple processors as pipelining would generally imply.

Notice, however, that parallelization of the above postloop also requires that the postloop be processed as an associative reduction.

2. Introduction

This paper will not attempt to describe the excellent work of [AlK82, All83, BuC86, Con86, Ell85, KAI85, KuS84, Li85, MiP86, PoK86, ScK86, and Wol86] in the parallelization of FORTRAN DO or similar loops; it is assumed that such technologies will be employed wherever appropriate. Our intent is instead to define a loop parallelization transformation to be used when the above techniques either do not readily apply or result in overly synchronization-intensive code. The class of loops for which such a new technique is needed is perhaps best summarized as those loops which cannot be parallelized in such way as to change the order of computational complexity. (Only parallelism in the form of executing multiple iterations of the loop body in parallel will be considered in this paper.)

If a sequential loop is of a certain form, then the loop's body for multiple iterations — parameterized in a way corresponding to the parameters of the body in the iterations — may be executed using asynchronous parallel control. The constraints on the loop form which permit such parallelization are, to some extent, dependent on the (dependence) analysis techniques used and also on the target machine characteristics (for example, on some machines, only certain memory reference patterns may be executed fully parallel). However, in this paper, we are more concerned with *defining the parameters of the body for each iteration* than with the data dependence and other constraints.

2.1. Iteration Parameters

What are the parameters of the execution of a loop body for a particular iteration? Any expression, computed within the loop body, whose value is not loop invariant[1] is potentially a parameter to the loop body. Further, the *existence* of a particular iteration is a parameter to the loop body. For the purposes of this paper, we define the expression which must be evaluated in order to determine if a particular iteration will occur as the **iteration decider**. Consider the following C code fragment:

```
for (i=0; i<n; ++i) {
    a[i] = 0;
}
```

Listing 3: Simple Parallelizable Loop

The loop body here may be considered as having parameters &a[i], i, and an iteration decider derived from the loop exit expression i<n. In practice, since &a[i] and i are related by a loop invariant expression (i = (&a[i])-(&a[0])), only one of them need be a parameter to the loop body.

2.2. Iteration Deciders

The iteration decider for the loop in Listing 3 is constructed as:

```
(0 + (k * 1)) < n
```

Listing 4: Iteration Decider for Listing 3

[1] An expression is loop invariant if its form does not change from one iteration to the next and if no values used in the expression may be changed by earlier iterations.

which was obtained by substituting the formula computing the value of `i` in iteration *k* (`0 + (`*k* `* 1))` within the loop exit test expression (`i<n`). Notice that this iteration decider is loop invariant; it can be determined that the *k*th iteration would or would not occur in the sequential execution of the loop without any dependence on a value computed in a previous iteration. In order for a loop to be large-grain parallelizable, it is necessary that the iteration decider be loop invariant.

It is also necessary that the iteration decider be defined for values of *k* as large as the number of iterations the sequential loop would execute plus the maximum number of parallel operations. This is due to the fact that the entire loop is asynchronous parallel *iff* each process is able to terminate itself, which implies that the iteration decider may be applied across all PEs despite the fact that as few as one may actually have work to do. Consider the loop:

```
for (i=0; a[i]<n; ++i) {
     a[i] = 0;
}
```

Listing 5: Loop with Exit Condition Hazards

here, the formula is:

```
a[0 + (k * 1)] < n
```

Listing 6: (Incorrect) Iteration Decider for Listing 5

which is loop-invariant, but it is not properly defined over the necessary range of values for *k*. It fails in two respects:

[1] Suppose `n` is `5`, the target machine is capable of executing four operations simultaneously, and the first four elements of the array `a[]` are `{ 0, 5, 3, 1 }`. The formula of Listing 6 would determine that iterations 0, 2, and 3 are to be executed, although the sequential loop would execute only iteration 0. In other words, this formula is not valid for any value of *k* larger than the number of iterations made by the sequential loop. An even simpler example of such a failure may be seen by replacing `i<n` of Listings 3 and 4 with `i!=n`.

[2] Suppose that the first element of `a` to satisfy the exit condition is the last element of the array. Ignoring, for the moment, that the formula of Listing 6 does not yield correct decisions for values of *k* larger than the number of iterations made by the sequential loop, it is clear that the test itself would cause a reference past the end of `a`.

Both of these kinds of failures can be corrected directly in some, but not all, cases:

- If the test in the loop was `i!=n`, it is within current compiler technology to convert the test into `i<n`. This transformation requires recognition of the linear sequence of values taken on by `i`, but this insight is also necessary in order to create a loop-invariant formula if the loop condition is `i<n`.
- If the test involves a subscripting operation, the compiler could automatically prefix the test with a bounds check for the array. If the subscript would be out of bounds, then the loop is terminated without performing the remainder of the loop control test. (Of course, this "fix" assumes that the sequential loop remained in bounds — which is not necessarily a good assumption.)

The vast majority of loops for which a proper iteration decider cannot easily be created are, however, loops which have been commonly characterized as either "inherently sequential" or as parallelizable only by "pipelining" (with no asynchronous parallelism). The asynchronous parallelization of this kind of loop by selectively serializing the synchronization-heavy portion of the loop is the aim of the current work.

Typical examples of loops for which only this kind of selective serialization can produce a proper iteration decider include:

- a loop which reads input from a single stream until some condition occurs in the stream (as in Listings 1 and 2),
- a loop which traverses a linked list, and
- a loop which is essentially a FORTRAN `DO` loop with an additional control-flow hazard (multiple entries or exits, embedded conditionals).

Many loops of these forms can provide substantial asynchronous parallelism using control precomputation. The basic idea is that each such loop can be converted into a sequential loop which computes the iteration deciders and an asynchronous parallel "loop" which is driven by the iteration deciders computed serially.

3. Terminology and Definitions

Since it is relatively convenient to do so, this discussion employs the terminology of conventional, sequential, compiler flow analysis, as per [AhU77 and AhS86]. These terms are **use**, **def**, **D-U chain**, and **U-D chain**. Two new terms, **def closure** and **use closure**, are also defined — but these are simply extensions of the concepts of D-U chains and U-D chains, respectively:

Definition 1: Def Closure (D*)

The **def closure**, denoted D*, of a particular def δ consists of the D-U chain of $\delta \cup$ the D-U chain of each def which establishes a binding where the value used in the binding was either in D* of δ or resulted from a computation involving at least one use that was a member of D* of δ. Informally, D* of δ consists of the def δ and the set of all subsequent uses and defs which may depend on prior execution of the def δ.

Definition 2: Use Closure (U*)

The **use closure**, denoted U*, of a particular use u consists of the U-D chain of $u \cup$ the U-D chain of each use which is either used to produce the value for a def which is an element of U* or is involved in a computation whose result is used to produce the value for a def which is an element of U*. Additionally, each use which is involved in selecting a flow path applied by D-U chains within U* of u is also a member of U* of u. Informally, U* of u consists of the use u and the set of all uses and defs which may have to be executed to produce the value which is used in use u.

These essentially embody the concept of closure of a D-U chain (for D*) and of closure of a U-D chain (for U*). No other terminology or definitions are needed.

4. Control Precomputation

The control precomputation transformation may be viewed as analysis and transformation of an arbitrary cycle within a flow graph, resulting in a functionally equivalent substitute for that cycle within the flow graph. In this view, it is convenient to consider the input to the algorithm as a subgraph of a conventional control flow graph (as defined above) which is known to be a cycle.

The algorithm does not require that the cycle in question be the result of structured code, nor even that it constitute a reducible graph. Given a reducible flow graph, the task of locating a cycle and marking its subgraph is easily accomplished. Although optimal parallelization might not result, irreducible graphs may be processed in like manner — by recognizing *any* apparent cycle. In such a case, any additional entries to or exits from the subgraph must be represented in the extracted subgraph: each cut point should be denoted by a dummy node in the subgraph.

In its most general form, the control precomputation transformation algorithm may be summarized as:

[1] **Straighten the flow graph.** A conventional control flow graph is constructed for the source program, but for each node A, if only a single arc leaves A and it goes to node B, and if no other arc ends in B, then nodes A and B are combined into a single node. (Since jumps are represented by arcs, each node should consist only of computational operations — no jumps are internal to a node.) Code straightening is discussed in [Die84].

[2] **Split loop by U* of exit expression(s).** Let P be the set of all uses which are in the exit condition expression(s) of the loop. For each operation o within the loop, if there exists a use $u \in P$ such that $o \in U^*(u)$, then o is placed in the preloop; otherwise, o is placed in the postloop. At this

point, it is possible to make a good estimate as to whether the loop can be usefully parallelized in this way — if not, the entire transformation can be aborted.

[3] **Promote defs across preloop/postloop.** For each def $\delta \in$ preloop, if $D^*(\delta)$ is not a subset of the preloop (i.e., a def in the preloop is used in the postloop), allocate a vector of temporary storage to buffer the value of this def between the preloop and the postloop. Adjust the references accordingly in the preloop and postloop.

[4] **Internal branch correction.** If the body of the source loop contained conditional branches, for each conditional that dominates code which is neither

- completely contained within the preloop nor
- completely contained within the postloop

create a vector of temporary storage to buffer the value of the conditional expression and replicate the conditional construct in both the preloop and the postloop. The preloop stores values into the vector and the postloop reads them, thereby duplicating the control flow which would have occurred within the sequential execution of each iteration.

[5] **Alternative entry correction.** If the source loop may be entered from points other than the top of the loop, only a portion of the first iteration may be executed, but all other iterations (except perhaps the last one) will execute the entire body just as though the loop had been entered at the top. Therefore, any jump which would enter the source loop becomes a jump to a chunk of code which replicates the portion of the first iteration which appears after the jump entry point. This chunk terminates with a jump to the test of the closure loop.

[6] **Alternative exit correction.** If the body of the source loop contained conditional exits at other than the edge between iterations, create a scalar state variable which may take on a value representing the exit condition which has occurred. (A scalar may be used since only the last iteration of the loop may be exited prematurely.) The body of the postloop is then replicated after the postloop, with the conditional clauses of the source loop replaced by references to the scalar state variable.

[7] **Construct the final looping structure.** The preloop, postloop, etc., are placed within a closure loop which permits buffer vectors to be of finite size (in Listing 2, of size `MAXWIDTH`). Typically, this size would be related to the parallelism width of the hardware. Parallelization of the postloop then proceeds by traditional methods.

Ignoring the complexity of the flow analysis and of later operations parallelizing the postloop, the complexity of this transformation is approximately $O(n)$, where n is the number of uses + defs within the loop.

5. A Complete Example

Due to space limitations, it is not possible to give examples of all the different types of loops which can be parallelized using the control precomputation transformation. Instead, this section presents in detail the parallelization of a loop which not only cannot be effectively parallelized by other means, but also embodies an irreducible flow graph. The C code appears in Listing 7. [illegible]

```
    for (i=0; a[i]<n; ++i) {
        a[i] = 0;
        b:
        c[i] = 0;
    }
```

Listing 7: Irreducible Source Loop with Control Dependencies

By rather conventional means, the source loop of Listing 7 can be converted into the tuple code of Table 1, generating the basic blocks listed in Table 2 and the flow graph of Figure 1.

Basic Block	**Set of Tuple #s**	**Exit Arc(s)**
A	{0}	B
B	{1, 2, 3, 4, 5, 6}	False(5) ⟶ C, else G
C	{7}	E
D	{8, 9, 10, 11}	B
E	{12, 13, 14}	F
F	{15, 16, 17, 18}	D
G	{19, . . .}	

Table 2: Grouping of Tuples into Basic Blocks

Tuple #	Operation	
0	`Store`	`#i,#0`
1	`Load`	`#i`
2	`Add`	`#a,1`
3	`Load`	`2`
4	`Load`	`#n`
5	`LT`	`3,4`
6	`JumpF`	`5,19`
7	`Jump`	`12`
8	`Load`	`#i`
9	`Add`	`8,#1`
10	`Store`	`#i,9`
11	`Jump`	`1`
12	`Load`	`#i`
13	`Add`	`#a,12`
14	`Store`	`13,#0`
15	`Load`	`#i`
16	`Add`	`#c,15`
17	`Store`	`16,#0`
18	`Jump`	`8`
19	`. . .`	

Table 1: Tuple Code Representing Listing 7

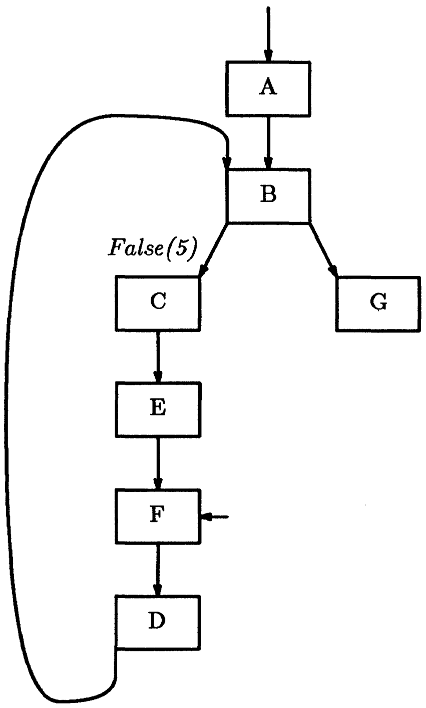


Figure 1: Original Control Flow Graph

This graph can be straightened (step 1 of the control precomputation algorithm) to create the graph in Figure 2 and the new set of basic blocks listed in Table 3.

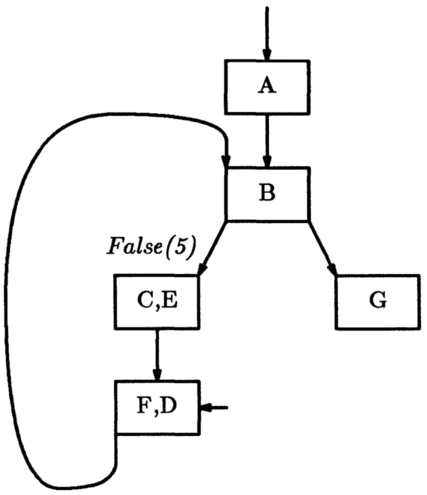


Figure 2: Straightened Control Flow Graph

Basic Block	Tuple Sequence	Exit Arc(s)
A	{0}	B
B	{1, 2, 3, 4, 5}	False(5) → {C, E}, else G
{C, E}	{12, 13, 14}	{F, D}
{F, D}	{15, 16, 17, 8, 9, 10}	B
G	{19, . . .}	

Table 3: Basic Blocks after Code Straightening

Next, correction is made for the multiple entry points to the loop. There is no particular reason to choose one entry over the other, however, trying both (see Figures 3 and 4) clearly demonstrates that the normalization for the entry at `for` is more efficient, resulting in the new set of basic blocks given in Table 4. Note that, since Figures 3 and 4 represent only the code derived from the loop body, in some cases *nil* nodes are introduced as placeholders for the entry and exit points where the loop body code was not replicated.

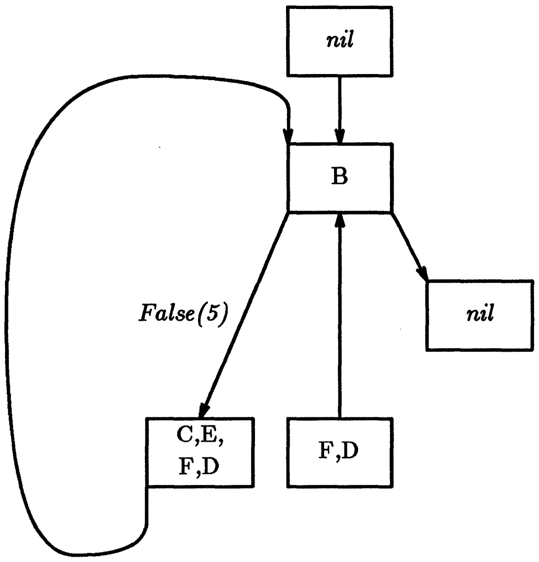


Figure 3: Normalized Graph for Entry at `for`

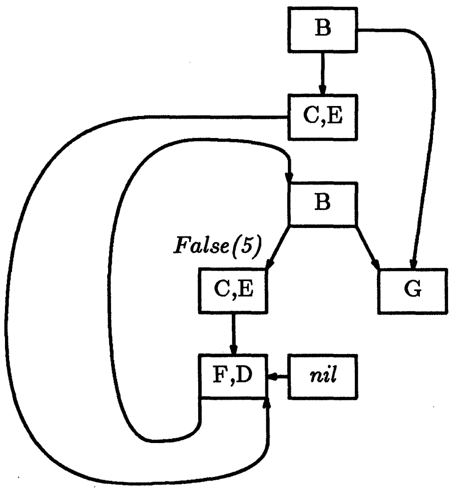


Figure 4: Normalized Graph for Entry at `b:`

Basic Block	Tuple Sequence	Exit Arc(s)
A	{0}	B
B	{1, 2, 3, 4, 5}	False(5) ⟶ C, E, else G
C, E, F, D	{12, 13, 14, 15, 16, 17, 8, 9, 10}	B
G	{19, . . .}	

Table 4: Basic Blocks after Normalization

Considering only those operations remaining in the loop (see Figure 5), the final loop structure can now be constructed using the U* and D* information in Table 5. The resulting preloop and postloop code is given in Table 6.

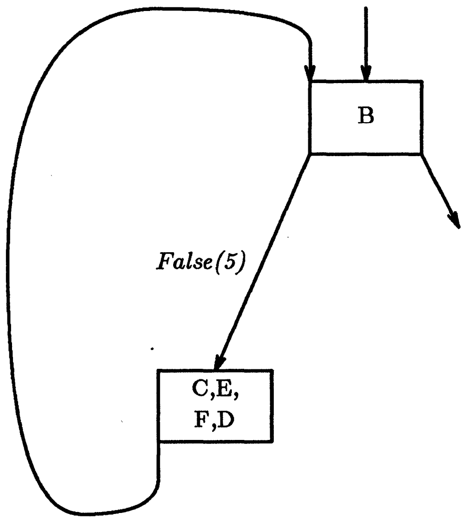


Figure 5: Extracted Normalized Loop

Tuple #	Operation		U*(5)	D*(10)
1	`Load`	`#i`	Yes	Yes
2	`Add`	`#a,1`	Yes	Yes
3	`Load`	`2`	Yes	Yes
4	`Load`	`#n`	Yes	No
5	`LT`	`3,4`	Yes	Yes
14	`Store`	`2,#0`	No	Yes
16	`Add`	`#c,1`	No	Yes
17	`Store`	`16,#0`	No	Yes
9	`Add`	`1,#1`	Yes	Yes
10	`Store`	`#i,9`	Yes	Yes

Table 5: Use/Def Closures for Optimized Graph

This parallelization permits the assignments `a[i] = 0;` and `c[i] = 0;` to be asynchronously parallelized for all iterations.

Block	Tuple #	Operation	
Preloop B	1	`Load`	`#i`
	2	`Add`	`#a,1`
		`Save`	*ASubIBuf*`,2`
	3	`Load`	`2`
	4	`Load`	`#n`
	5	`LT`	`3,4`
Preloop C, E, F, D	9	`Add`	`1,#1`
	10	`Store`	`#i,9`
		`Save`	*IBuf*`,2`
Postloop body	*X*	`Restore`	*ASubIBuf*`,2`
	14	`Store`	*X*`,#0`
	Y	`Restore`	*IBuf*`,2`
	16	`Add`	`#c,`*Y*
	17	`Store`	`16,#0`

Table 6: Resulting Preloop/Postloop Contents

6. Summary

The principle of parallelization by selective serialization is, as evidenced by the control precomputation transformation, a very effective way in which to minimize synchronization overhead — thereby maximizing asynchronous parallelism. This is vital in automatic parallelization for non-shared memory MIMD machines, such as the current generation of hypercube computers.

We have not yet determined how often such parallelization opportunities arise in real programs, however, it is our belief that the high frequency of occurrence of these loops in C programs we have examined is not anomalous. While FORTRAN number-crunching codes often employ easy-to-parallelize loops, the nature of systems programs, typical of C code, is such that unpredictable `whiles` appear to far outnumber `DO`-like loops. Further study is underway to confirm or deny this.

References

[AhS86] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, Massachusetts, 1986.

[AhU77] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison Wesley, Reading, Massachusetts, 1977.

[AlK82] J. R. Allen, K. Kennedy, "PFC: A Program to Convert Fortran to Parallel Form," Department of Mathematical Sciences, Rice University, Houston, Report MASC TR 82-6, March 1982.

[All83] J. R. Allen, *Dependence Analysis for Subscripted Variables and its Application to Program Transformations*, Rice University, Ph.D. Thesis, April 1983.

[BuC86] M. Burke, R. Cytron, "Interprocedural Dependence Analysis and Parallelization," SIGPLAN Symposium on Compiler Construction, 1986, pages 162-175.

[Con86] *Vectorizing C Compiler*, Convex Computer Corporation, Richardson, Texas, 1986.

[Die84] H. G. Dietz, *Compiler Design and Construction II*, Graduate Course Notes, Polytechnic Institute of New York, Spring 1984.

[Ell85] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, ACM Doctoral Dissertation Award, MIT Press, 1985.

[KAI85] "Mini-KAP/AF," Kuck and Associates, Inc., new product release, 1985.

[KuS84] D. J. Kuck, A. H. Sameb, R. Cytron, A. V. Veidenbaum, C. D. Polychronopoulos, G. Lee, T. McDaniel, B. R. Leasure, C. Beckman, J. R. B. Davies, and C. P. Kruskal, "The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance," IEEE Proceedings of the 1984 International Conference on Parallel Processing, August 1984.

[Li85] Z. Li, *A Technique for Reducing Data Synchronization in Multiprocessed Loops,* MS Thesis, University of Illinois at Urbana-Champaign, May 1985.

[MiP86] S. P. Midkiff and D. A. Padua, *Compiler Generated Synchronization for DO Loops,* Technical Report, University of Illinois at Urbana-Champaign, Number CSRD 554, 1986.

[PoK86] C. D. Polychronopoulos, D. J. Kuck, and D. A. Padua, *Execution of Parallel Loops on Parallel Processor Systems,* University of Illinois at Urbana-Champaign, Number CSRD 552, 1986.

[ScK86] R. G. Scarborough and H. G. Kolsky, "A Vectorizing Fortran Compiler," IBM Journal of Research and Development, Volume 30, Number 2, March 1986.

[Wol86] M. Wolfe, "Advanced Loop Interchanging," preprint extended version of a paper appearing in the Proceedings of the 1986 International Conference on Parallel Processing, 1986.

MINIMIZING COMMUNICATION FOR SYNCHRONIZING PARALLEL DATAFLOW PROGRAMS*

Lee Badger
Computer Science Department
University of Maryland, College Park
College Park, MD, 20742, USA
and
Mark Weiser
Computer Sciences Laboratory
Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA 94304

Abstract –A new method of automatically parallelizing sequentially written programs is to use dataflow information to break programs into components which execute concurrently and without communication. At execution time, the output streams of the separate components are merged to construct the sequential behavior of the parallelized program. This paper presents a technique to perform the merging operation with approximately minimal communication. The merging algorithm is potentially applicable to any parallel computer which uses dataflow paths to organize the computation.

Introduction

This paper describes an algorithm which substantially improves on a new way to automatically parallelize existing sequential programs. Traditional parallelism extracting approaches include pipelining [5], vectorizing [1], [2], and the development of dataflow techniques [4]. Pipelining seeks to keep many instructions simultaneously in different stages of execution. Vectorizing techniques attempt to execute all cycles of a loop simultaneously. Dataflow machines attempt to execute as many fine grained data manipulation operations as are enabled at any time. These techniques sometimes achieve significant speedup [6], but usually require special hardware or the tight coupling of processors [6], [8]. We propose a general technique which can tolerate slow interprocessor communications and which is therefore applicable for use in a wide variety of MIMD [7] environments. Our method is described in terms of program slicing [12], although it has potential for application to many parallelization techniques based on dataflow.

Slicing is a technique for breaking a program into dataflow-related components called slices. The slices are distributed onto separate processors and executed concurrently. The slices are themselves complete programs, and normal vectorizing techniques such as scalar renaming, variable expansion, node splitting, loop distribution, etc. [9], [11] can be used to increase their execution speeds. Their output streams are collected at a central location and assembled at runtime to form the output of the original program. The main contribution of this paper is an efficient way to perform this last process, which we call splicing.

A slice of a program P is a subprogram of P which has the property that it faithfully reproduces some portion of P's behavior. For instance, if P contains an output statement o, then a slice P_i of P, which contains o, at runtime executes o exactly the same number of times that P does (on the same input), and the i'th execution of o

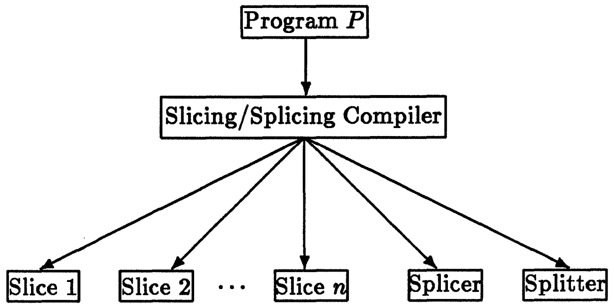


Figure 1: Compile Time

in P_i produces exactly the same value in the output as the i'th execution of o in P. A slice need not contain statements which compute values which are not output in that slice. If P contains k output statements, then we may generate k slices of P, $P_1, P_2, \ldots, P_k$, such that the i'th slice contains the i'th output statement and no other output statement. These slices, along with modules to split their common input and merge their output, may be generated automatically by a slicing/splicing compiler (see figure 1). Executing concurrently and without communication, the slices reproduce all of P's behavior (see figure 2), but the arbitrary differences in the speeds of individual slices may cause the output to be permuted. The ***splicing problem*** is the problem of finding at runtime the correct ordering of the output.[a] [b]

Previous solutions to the splicing problem exist [10], [13], but they require each P_i to continually transmit to a central location a sequence of nodes or edges which represents P_i's walk in its flowgraph. These solutions pose efficiency problems because the amount of information transmitted by each slice is proportional to the running time of the slice. Our new solution is an improvement because it allows each slice to transmit only a small amount of information along with each piece of normal output, thus

[a] A symmetric problem, which has been dubbed "splitting", concerns how slices read from a common input file. The mechanism which we develop here to order write operations, however, may be applied to order read operations, and so we address the splitting problem only implicitly.
[b] For the reader who may not be familiar with slicing and splicing it is important to note that figures 1 and 2 oversimplify in two ways. First, the complete program is shown being sliced as a whole, when in fact slicing and splicing are applicable to any single-entry-single-exit program component and one may wish to choose components for maximum parallelism. Second, it shows only a single splicer, while in fact splicers can be cascaded to reduce bandwidth and fan-in.

* This research was supported in part by ONR grant N00014-87-K-0307 and by Xerox Corp.

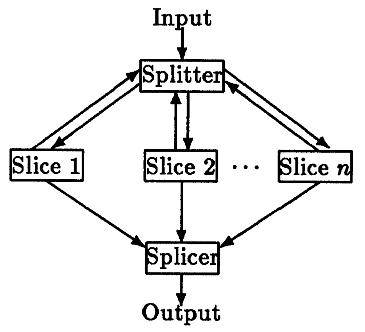


Figure 2: Run Time Communication

practically eliminating the overhead associated with splicing. In fact, we argue later that the amount of information we transfer is close to a lower bound.

Algorithm

In the new solution, a small collection of counting variables is added to each slice to record information about its progress during execution. When each slice encounters an output statement, it sends to the splicer module both the output that it has produced and the sum of its counting variables. (By the properties of slices, different outputs from the same slice are produced in the correct order, and we assume for simplicity that the communication medium is FIFO.) When the splicer receives two (output, counter-sum) pairs from two different slices, it must decide how to order the outputs produced. For some outputs, a static analysis performed when the program was first sliced provides enough information for the splicer to decide which output should appear first in the program's execution. For other outputs a static analysis is insufficient, and the counters are carefully chosen at slice-time to provide the additional information to resolve these cases at runtime.

Let P be the original program to be parallelized. Without loss of generality, we will assume throughout this paper that there are only two slices of P, P_i and P_j, and only two output statements in P, o_i and o_j, such that o_i appears only in P_i and o_j appears only in P_j. For brevity we will use P_k and o_k to denote an arbitrary slice or output statement. We now present some definitions which are necessary to express the algorithms which preprocess P, generate the counting variables for each slice, and perform the splicing.

A ***hammock graph*** is a structure (N, E, n_o, n_e) such that N is a set of nodes, E is a set of edges between elements of N, and n_o is a start node and n_e is an end node such that every element of N can be included in some directed path p from n_o to n_e. For our purposes, we assume that $outdegree(n_e) = 0$. (Note that any hammock can easily be augmented to make this true, and that any flowgraph can easily be augmented to be a hammock.) We assume that program P's flowgraph G is such a hammock. The flowgraph of slice P_k is also a hammock and we denote its set of nodes N_k and note that $N_k \subseteq N$. For notational simplicity we assume that $outdegree(o_k) = indegree(o_k) = 1$. (Our results do not depend in any essential way on this simplification.) Define an inverse dominator of a node n to be another node n' such that every path from n to n_e contains n'. Denote a branch node ($outdegree > 1$) by b. For any node n in G, define $I(n) = \{b | n$ is on some directed path p from b to b's nearest inverse dominator such that p contains b's nearest inverse dominator only once, and n is not on an endpoint of $p\}$. $I(n)$ is the set of branch nodes which can influence the execution of n [3]. Let $I = I(o_i) \cap I(o_j)$. I is thus the set of branch nodes which can affect the execution of both o_i *and* o_j.

The key idea of our method can be stated as follows: Only statements in I can affect the relative ordering of o_i and o_j, so to synchronize o_i and o_j it suffices to know where, relative to the number of executions of statements in I, each slice was when it produced o_i (Theorem 1). Furthermore, if both slices are at the same place relative to executions of statements in I, then the relative order of o_i and o_j is completely determined, and the same, for all cycles in the flowgraph (Theorem 2). In order to better motivate the proofs of these theorems, below are brief descriptions of the algorithms to preprocess P, to generate the tags which accompany the outputs of slices, and to reconstruct P's intended output at the splicer module. The correctness of these algorithms depend on the theorems proved in the following section.

Preprocessing P

Find at compile time a directed path p in G which connects o_i and o_j and contains no element of I, if such a path exists. If p exists, then we remember its direction: if p is from o_i to o_j, then we remember that o_i takes precedence over o_j. The direction of p, if it exists, provides the extra information which, when combined with the values of the counting variables associated with o_i and o_j, allows the splicer to correctly order an occurrance of o_i and an occurrance of o_j in the output. If p does not exist, then we will not need this information.

Tag generation

In P_i and P_j, for each element of I, initialize a counter variable to the value 0 before execution begins. Each time execution passes a node in I, increment the corresponding counter variable. When an output statement o_i or o_j is encountered, send the collection of counters, along with the value to be output, to the splicer. We refer to this collection of counters as a tag.

Splicing algorithm

At runtime, order two outputs (from different slices) by comparing their tags. The tag whose counters add to a larger sum is larger. The output statement with the smaller tag preceeds the output statement with the larger tag. If the tags are equal, then the output statement with the greater precedence (from the preprocessing of P) preceeds the other.

Proof of Correctness

We model the execution of P by a walk $w = \langle n_o = n_1, n_2, n_3 \ldots, n_n = n_e \rangle$ where each n_i is an occurrance of a node in G. Denote a prefix of w as $\tilde{w}$. A special prefix of a walk in a slice which occurs quite often is the prefix which ends at the output statement. We usually refer to this prefix, for slice k and statement o_k, as simply $\tilde{w}_k$. For $A \subseteq N$ let $w_{|A}$ be the sequence w with every $n_i \notin A$ removed. Let $|w|$ denote the length of w, and let $||$ denote concatenation between sequences. If we let slice P_i have flowgraph G_i then we model the execution of P_i by a walk w_i in G_i. According to the definition of a slice we must have $w_i = w_{|N_i}$ and $w_{i|\{o_i\}} = w_{|\{o_i\}}$. In order for P_i to generate

$w_{i|\{o_i\}}$, (in other words, in order for P_i to be a slice) P_i must contain every element of I, and the elements of I must behave in P_i exactly as they do in P. We thus have $w_{|I} = w_{i|I}$ and, analogously, $w_{|I} = w_{j|I}$. Note that $\tilde{w}_{i|I}$ and $\tilde{w}_{j|I}$ are both prefixes of the same sequence, $w_{|I}$. During execution, when slice P_i reaches execution state $\tilde{w}_i$ such that the last element of $\tilde{w}_i$ is the output statement o_i, it sends to the splicer the result of o_i and also the sum of its set of counters, which is equal to $|\tilde{w}_{i|I}|$.

The splicer module will receive and compare $|\tilde{w}_{i|I}|$ and $|\tilde{w}_{j|I}|$. They will either be unequal or equal. We will consider both cases and show that in each case the splicer is able to decide which output should preceed the other in w. We first consider the case where they are unequal. Let the irreflexive, asymmetric, transitive relation $n_i \prec_s n_j$ denote that n_i preceeds n_j in sequence s and let $last(s)$ denote the last element of s. We wish to prove:

$$|\tilde{w}_{i|I}| < |\tilde{w}_{j|I}| \Rightarrow last(\tilde{w}_i) \prec_w last(\tilde{w}_j)$$

This assertion states that, if the tags are unequal, then the output of the slice which sent the smaller tag should preceed the output of the slice which sent the larger tag. In order to prove this assertion, we require two technical lemmas.

The following lemma asserts that, if two elements of a projected sequence are related by $\prec$, then they are also related in the sequence on which the projection is defined.

Lemma 1

For all $s \in N^+$, $A \subseteq N$, and n, m elements of $s_{|A}$

$$n \prec_{s_{|A}} m \Rightarrow n \prec_s m$$

Proof: We show it by induction. Let x be an element of $N - A$. We have $s_{|A} = s_1||\langle n\rangle||s_2||\langle m\rangle||s_3$ where s_1, s_2, and s_3 are subsequences of $s_{|A}$. Now let $A' = A \cup \{x\}$. We now have $s_{|A'} = s'_1||\langle n\rangle||s'_2||\langle m\rangle||s'_3$ where s'_i is s_i with possibly many instances of x inserted in it. By transitivity the relation is preserved. QED.

The next lemma displays a necessary relationship between elements of a projection of a prefix of a sequence which are in the projection and elements of the prefix which are not in the projection. Let $[s]_i$ denote the i^{th} element of sequence s.

Lemma 2

For all $s \in N^+$, $A \subseteq N$, and $v \in$ Integers

$$|\tilde{s}_{|A}| < v \leq |s_{|A}| \Rightarrow last(\tilde{s}) \prec_s [s_{|A}]_v$$

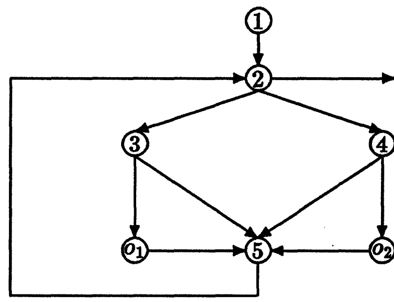
Proof: We prove it by contradiction. Suppose that $[s_{|A}]_v \prec_s last(\tilde{s})$. Then $[s_{|A}]_v$ must be in $\tilde{s}_{|A}$. Let $u = |\tilde{s}_{|A}|$. Then $last(\tilde{s}_{|A}) = [s_{|A}]_u$ and we must have $u \geq v$, but by the left hand side we have $u < v$, a contradiction. QED.

We are now able to prove that the splicer's decisions are correct when the tags are unequal.

Theorem 1

$$|\tilde{w}_{i|I}| < |\tilde{w}_{j|I}| \Rightarrow last(\tilde{w}_i) \prec_w last(\tilde{w}_j)$$

Proof: Informally this must be approximately true, since w_j is "farther ahead" than w_i in executing statements in



$w = \langle 1,2,3,o_1,5,2,4,o_2,5,2,6\rangle$ $I = \{2\}$
$\tilde{w}_1 = \langle 1,2,3,o_1\rangle$ $\tilde{w}_{1|I} = \langle 2\rangle$ $|\tilde{w}_{1|I}| = 1$
$\tilde{w}_2 = \langle 1,2,3,o_1,5,2,4,o_2\rangle$ $\tilde{w}_{2|I} = \langle 2,2\rangle$ $|\tilde{w}_{2|I}| = 2$

Figure 3: Example when projections are not equal.

I, and only statements in I matter for relative ordering of o_i and o_j. Lemmas 1 and 2 permit us to generalize from prefixes of walks projected through I (which is the sort of information we get from $|\tilde{w}_{k|I}|$) to prefixes of the base walks. Formally, we will show that, for some $x \in w$, $last(\tilde{w}_i) \prec_w x$ and $x \prec_w last(\tilde{w}_j)$. Let $v = |\tilde{w}_{j|I}|$ and notice that $v \leq |w_{j|I}| = |w_{i|I}|$ and that $|\tilde{w}_{i|I}| < v$. We now have:

$$|\tilde{w}_{i|I}| < v \leq |w_{i|I}|$$

and by lemma 2 we have:

$$last(\tilde{w}_i) \prec_{w_i} [w_{i|I}]_v$$

Using the equality $w_{|N_i} = w_i$ we have:

$$last(\tilde{w}_i) \prec_{w_{|N_i}} [w_{i|I}]_v$$

Applying lemma 1 we get

$$last(\tilde{w}_i) \prec_w [w_{i|I}]_v$$

Now notice that $[w_{i|I}]_v = [w_{j|I}]_v$ and that $[w_{j|I}]_v = last(\tilde{w}_{j|I})$. Remember that $last(\tilde{w}_j)$ is an output statement, not an element of I, and therefore:

$$last(\tilde{w}_{j|I}) \prec_{w_j} last(\tilde{w}_j)$$

Substituting $[w_{i|I}]_v$ back for $last(\tilde{w}_{j|I})$ we have:

$$[w_{i|I}]_v \prec_{w_j} last(\tilde{w}_j)$$

Using the identity $w_j = w_{|N_j}$ and lemma 1 we have:

$$[w_{i|I}]_v \prec_w last(\tilde{w}_j)$$

Combining previous results:

$$last(\tilde{w}_i) \prec_w [w_{i|I}]_v \prec_w last(\tilde{w}_j)$$

and by transitivity of $\prec_w$ we have:

$$last(\tilde{w}_i) \prec_w last(\tilde{w}_j)$$

which is our desired result. QED.

Theorem 1 enables the splicer to make the correct demore, rather than transmitting the complete counter value each time, each slice could simply transmit the increment in the counter value since the last time it was transmitted, and the splicer could keep track of the full value. Finally, an upper bound (e.g. 16 bits) could be set on the size of the integer representing the counter increment. Whenever the upper bound was in danger of being exceeded the slice would transmit an additional increment to the slicer.

Is transmitting the sum of the number of times the statements in I have been executed the least amount of information one can get away with for synchronizing the slices? And is incrementing a counter for each statement in I the least amount of execution overhead possible in the slices? At this time we do not have a formal answer to these questions. However, we believe our method to be optimal, or close to it. Our thinking runs like this: Consider an inner loop like the one shown in figure 4. o_i and o_j can each be avoided an arbitrary number of times in their respective slices, allowing one slice to be arbitrarily far ahead of the other before producing output requiring synchronization. The only way to synchronize at that point is to know where each slice is in its computation, which requires at least a count of the number of times each slice went around its inner loop, and the size and expense of this count is equivalent to the size and expense of the counters required for our algorithm.

To conclude, we have described a method to parallelize sequentially written programs. Program slicing, combined with program splicing, provides a way to concurrently execute parallelized programs on loosely coupled multiprocessors. We then introduced a new method for resynchronzing the outputs of slices which radically reduces the overhead associated with reconstruction of the output. We believe that program slicing, augmented with program splicing, holds out significant potential for the automatic parallelization of sequentially written programs. Work is currently proceeding on reducing the complexity of finding minimal tags, on implementing a slicing/splicing compiler, and on applying the splicing method to other dataflow-based parallelisms besides slicing.

Acknowledgments

The authors would like to thank Michael Mazurek for suggestions on earlier versions of this paper.

References

[1] G. H. Barnes, R. M. Brown, M.Kato, D. J.Kuck, D. L.Slotnick and R. A. Stokes, "The ILLIAC IV Computer," IEEE Transactions on Computers (August,1968), pp. 746–757.

[2] K. Batcher, "STARAN Parallel Processor System Hardware," 1974 National Computer Conference And Exposition (May, 1974), pp. 405–410.

[3] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," Communications of the ACM (July, 1977), pp. 504–513.

[4] J. Dennis, "The Varieties of Data Flow Computers," First International Conference on Distributed Computing Systems (October, 1979), pp. 430–439.

[5] J. Dennis and G. Rong, "Maximum Pipelining of Array Operations on Static Data Flow Machines," Proceedings of the 1983 International Conference on Parallel Processing (August,1983), pp. 331–334.

[6] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, "Parallel Processing: A Smart Compiler and a Dumb Machine," Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Contruction (June, 1984), pp. 37–47

[7] M. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Transactions on Computers (September, 1972), pp. 948–960.

[8] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," Eighth ACM Symposium on Principles of Programming Languages, (January, 1981), pp. 207-218.

[9] D. J. Kuck, R. H. Kuhn, B. Leasure and M. Wolfe, "The Structure of an Advanced Vectorizer For Pipelined Processors," Fourth International Computer Software And Applications Conference, (October, 1980), pp. 709–715.

[10] M. Mazurek, M. Weiser, "Towards the Automatic Parallelization of Sequential Programs," Submitted for publication.

[11] D. A. Padua, M. J. Wolfe, "Advanced Compiler Optimizations For Supercomputers," Communications of the ACM (December, 1986), pp. 1184–1201.

[12] M. Weiser. "Program Slicing," IEEE Transactions On Software Engineering (July, 1984), pp. 352–357.

[13] M. Weiser. "Reconstructing Sequential Behavior From Parallel Behavior Projections," Information Processing Letters (October, 1983), pp. 129–135.

cision when tags are not equal. Figure 3 shows an example flowgraph and execution (w) where $|\tilde{w}_{1|I}| < |\tilde{w}_{2|I}|$ which implies $|\tilde{w}_1| < |\tilde{w}_2|$ which implies $last(\tilde{w}_1) \prec_w last(\tilde{w}_2)$. The main result of this paper is theorem 2, which enables the splicer to make the right decision when the tags are equal. Before presenting it, however, it is useful to introduce the following lemma, which asserts that every cycle in a flowgraph contains at least one branch node which influences the execution of every node in the cycle.

Lemma 3

In a hammock G, for every cycle $c = \langle n_1, n_2, n_3 \ldots, n_m, n_1\rangle$

$$\exists_{i,1\leq i\leq m}\forall_{j,1\leq j\leq m}\,[n_i \in I(n_j)]$$

Proof: Informally, this follows from the observation that there must be a branch node with an edge on a direct (off the cycle and itself cycle-free) path to n_e, and this node influences every node in the cycle because it can prevent any further executions of every node in the cycle. Formally, because G is a hammock, n_e must be reachable from every node in c and therefore at least one node in c has $outdegree > 1$. Assume for a moment that there is only one such node, and denote it b. At least one of b's outedges must be directed to a node n' which is not in c and which is on a cycle free path p from b to n_e. Since b's nearest inverse dominator is distinct from b and is on any cycle free path from b to n_e, b' nearest inverse dominator is on any cycle free path $\tilde{p}$ from a successor of b to n_e. Thus the inverse dominator of b is not in c. Now note that, since b in on c, c may be rewritten $\bar{c} = \langle b = n_k, n_{k+1}, n_{k+2}, \ldots, n_{k \bmod m+1}, n_1, \ldots, b\rangle$. The path $\bar{c}||\tilde{p}$ includes every node in c on a path from b to b's nearest inverse dominator and includes b's nearest inverse dominator only once (because $\tilde{p}$ contains no cycles). Thus $b \in I(n_i)$ for $1 \leq i \leq m$. If there are additional branch nodes which leave the cycle, they do not affect the existence of $\bar{c}||\tilde{p}$, and so b still influences every node in the cycle. QED.

The following result shows that, whenever tags are equal, the information from the preprocessing algorithm enables the splicer to decide which output should preceed the other. Intuitively, Theorem 2 asserts that, for every program P with flowgraph G, and any two slices P_i and P_j of P, exactly one of two things is true: in every possible walk w in G (execution of P) $last(\tilde{w}_i) \prec_w last(\tilde{w}_j)$ whenever their tags are equal, or in every possible walk w in G $last(\tilde{w}_j) \prec_w last(\tilde{w}_i)$ whenever their tags are equal. Figure 4 shows an example flowgraph and execution (w) where the two tags $|\tilde{w}_{1|I}|$ and $|\tilde{w}_{2|I}|$ are equal and where the path (p) found during the preprocessing algorithm allows the splicer to conclude that $last(\tilde{w}_1) \prec_w last(\tilde{w}_2)$. Let $w \in G$ mean that w is a walk in hammock G.

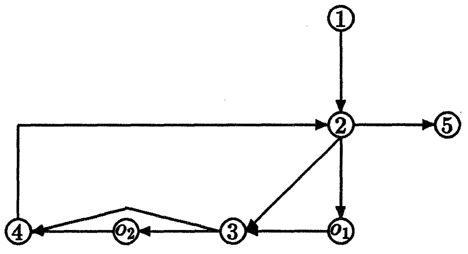
Theorem 2

For o_i and o_j distinct

$$\forall_G[\forall_{w\in G}\{|\tilde{w}_{i|I}| = |\tilde{w}_{j|I}| \Rightarrow last(\tilde{w}_i) \prec_w last(\tilde{w}_j)\} or$$

$$\forall_{w\in G}\{|\tilde{w}_{i|I}| = |\tilde{w}_{j|I}| \Rightarrow last(\tilde{w}_j) \prec_w last(\tilde{w}_i)\}]$$

Proof: First we show that, for any w, if $|\tilde{w}_{i|I}| = |\tilde{w}_{j|I}|$ then there exists at least one directed path p in G such that p contains o_i and o_j and p contains no element of I. Since the prefixes $\tilde{w}_i$ and $\tilde{w}_j$ end in different nodes, they cannot be of the same length. Let s denote the shorter prefix, l denote the longer, and d denote the difference so that $l = s||d$. Note



$w = \langle 1,2,o_1,3,o_2,4,2,5\rangle$ $I = \{2\}$ $p = \langle o_1, 3, o_2\rangle$
$\tilde{w}_1 = \langle 1,2,o_1\rangle$ $\tilde{w}_{1|I} = \langle 2\rangle$ $|\tilde{w}_{1|I}| = 1$
$\tilde{w}_2 = \langle 1,2,o_1,3,o_2\rangle$ $\tilde{w}_{2|I} = \langle 2\rangle$ $|\tilde{w}_{2|I}| = 1$
Because of p, o_1 preceeds o_2 whenever $|\tilde{w}_{1|I}| = |\tilde{w}_{2|I}|$.

Figure 4: Example when projections *are* equal.

that $|\tilde{w}_{i|I}| = |\tilde{w}_{j|I}| \Rightarrow |d_{|I}| = 0$ and d has no element of I. Now let o_S be the last node of s, and let o_L be the last node of l. Note that $o_S \notin I$ because every element of I is a branch node, and o_i and o_j are not branch nodes. Therefore the path $p = \langle o_S\rangle||d$ contains o_i and o_j and no element of I. If all such paths are directed in the same direction, that is, all are from o_i to o_j, or all are from o_j to o_i, then the ordering of o_i and o_j is always the same, for a particular graph, whenever the tags are equal. This is because any contrary ordering would require that d contain some element of I, and the tags would therefore be unequal. We now, then, must show that there exists no directed path p' of the form $\langle o_L, e, n, e, \ldots, o_S\rangle$ such that p' contains no element of I. We show it by contradiction. Suppose such a p' existed. Then, if we remove the last node of p to obtain $\bar{p}$, $\bar{p}||p'$ is a cycle C which contains both o_i and o_j. From lemma 3 we know that C must contain at least one element of I and therefore $|C_{|I}| \neq 0$, a contradiction. All paths which connect o_i and o_j and contain no element of I therefore have the same direction. QED.

Applications and Conclusion

Slices and splices are a convenient way for describing this sort of dataflow problem because the dataflow-independent pieces are separated into different programs. However, our methods are more generally applicable to dataflow-driven solutions. Consider for a moment a dataflow machine architecture which has compiled a sequential program into a number of small operations executed in parallel as their data becomes available. Following a single dataflow thread through such a machine shows a computation much like a walk through a slice, except that the parallelism is more constrained by later operations needing to wait for earlier ones to reach the same control point before proceeding. By using a splicing-style synchronization, faster computation threads waiting only for control synchronization could proceed without waiting, and later be merged by using the results of theorems 1 and 2. By removing a dependency, this could lead to better balancing of the execution units of the dataflow machine by keeping more of them busy.

There are some simple transformations of the algorithms described in previous sections which would make them more useful in practice. Rather than keeping a separate counter for each statement in I, a single counter incremented by each member of I could be used. Further-

Semi-Static Dataflow

Bruno R. Preiss

Department of Electrical Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1

V. Carl Hamacher

Computer Systems Research Institute
University of Toronto
Toronto, Ontario, Canada, M5S 1A4

Abstract

In this paper we present a new dataflow execution model called semi-static dataflow. This model incorporates aspects of conventional static and dynamic dataflow architectures. Programs are partitioned into a collection of dataflow graphs. The execution of each of these graphs is the responsibility of a low-level process called a *context.* The static dataflow execution model is used to evaluate each of these graphs. Separate instruction and data spaces are used to allow program reentrancy. Function invocation, iteration, and conditional execution are accomplished by dynamically creating new contexts.

The process of creating new contexts and moving data tokens between contexts is called *dynamic dataflow graph splicing* and is the motivation for calling the whole system semi-static. We present a number of programming paradigms for function invocation, sequential iteration and parallel iteration that are based on dynamic dataflow graph splicing.

We have simulated the execution of a semi-static dataflow multiprocessor. In this paper some of the simulation results obtained for several benchmark programs are presented.

1. Introduction and Motivation

Dataflow architectures are classified as either static or dynamic. Static and dynamic architectures differ in the following ways: First, in static architectures, the program graph is loaded into memory in completed form before the program begins execution (i.e., no run-time loader); whereas in dynamic architectures, nodes can be created at run time (e.g., to support loop unravelling and recursion). Second, in static architectures, at most one instance of an actor may be enabled for firing at a time. Dynamic architectures support several instances of an actor firing simultaneously[1]. Finally, static architectures use the same storage space for instructions (actors) and data (tokens) (i.e., impure code). Dynamic architectures use physically separate memories for instructions (i.e., pure code) and data[2].

In static dataflow architectures, the dataflow program graph is represented as a multiply-linked collection of items called *activity templates*[3]. Each activity template corresponds to an actor of the dataflow program graph. An activity template is a triple consisting of i) an operation code, ii) a set of operand slots, and iii) a (destination) pointer list. The operand slots correspond to the input arcs of the actor. Each slot is a reserved memory location into which one of the actor's predecessors stores a token. Thus, arcs have a maximum token-carrying capacity of one. The elements of the destination pointer list correspond to the output arcs of the actor. These pointers indicate the operand slots in other activity templates into which the result of this actor is to be stored. The operation code field of the actor specifies the function computed by the actor. It also directly or implicitly specifies the number of operand slots, the number of destination pointers, and the firing rule for the given actor. The basic execution cycle involves: identifying an activity template whose firing rule is satisfied (i.e., all its operand slots are full and the operand slots of its successors are empty); computing the function specified by the operation code; and storing the result in the operand slots specified by the destination pointers.

The finite arc capacity and impurity of code in the static dataflow systems result in dataflow program graphs that are not reentrant. This constraint affects both iteration and function (subroutine) calling mechanisms. Automatic run-time loop unravelling is not supported by static dataflow systems. Functions can be made reentrant by either code-duplication [4] or code-sharing mechanisms[5].

In dynamic dataflow architectures, dataflow program graphs are also represented as a collection of multiply-linked activity templates. In this case, activity templates consist of i) an operation code and ii) a (destination) pointer list. No space is reserved in the activity template for storing (input) data tokens. Instead, tokens are stored elsewhere. Each token is tagged with information that identifies the arc on which the token conceptually resides. Thus, a token consists of a data field and a tag field. As a result, the token-carrying capacity of the arcs is not constrained. As in static architectures, the elements of the destination pointer list correspond to the output arcs of the actor. These pointers specify the address of the successor activity template and also indicate to which input of the actor the token is to be sent. (In general, the functions computed by the actors need not be commutative). In addition, the pointer may also specify how many tokens are required by the successor actor in order to fire (to facilitate efficient evaluation of its firing rules). The basic execution cycle involves locating actors whose input arcs are full (by matching tag fields of tokens); computing the function specified by the operation code (using the value fields of tokens); and forming the value parts of the output tokens from the result of the computation and the tag parts of the output tokens from the elements of the destination list.

Since there are no operand slots in the activity templates, the code is pure. Furthermore, as explained above, the token-carrying capacity of arcs is not constrained by the representation of the dataflow graph. As a result, loop unravelling and reentrant functions are easily implemented in dynamic dataflow architectures.

To support reentrancy and unravelling, the tag field of tokens is augmented with information that identifies the context in which the token is executing. The augmented tag field is called an *activity name.* The activity name specifies i)

This research was supported in part by NSERC (Canada) under Grant A5192.

the actor (and which of its inputs) to which the token is destined, ii) the iteration number of the loop to which the token belongs (if it is inside a loop), and iii) the activity name of the calling function if the token belongs to a called function[6]. Note that the activity name is recursively defined. In effect, every token carries with it information that is analogous to the processor stack in conventional, Von Neumann architectures.

Iteration and function calls are accomplished in dynamic architectures by using actors that manipulate the tag fields of tokens[7]. For example, iteration is accomplished using an actor that takes tokens from the bottom of a loop, increments the iteration number field of the activity name of the token, and injects the token back into the top of the loop. Arguments are transmitted to functions and results are received from functions using actors that, in effect, push and pop contexts from activity names.

In this paper we present a new dataflow execution model that attempts to exploit the capabilities of both static and dynamic dataflow architectures without incurring the the tag matching overhead of the dynamic model or the difficulties arising from the impure code of the static model. We call this execution model semi-static dataflow. In semi-static dataflow architectures, the dataflow program graph is represented as a multiply-linked collection of activity templates. As in static architectures, the activity templates consist of i) an operation code part, ii) operand slots, and iii) destination pointers. Semi-static architectures differ from static ones in that the activity template is not stored in contiguous memory locations. Instead, the operation code and destination pointers are stored in an instruction space and the operand slots are stored in a data space. By associating several data spaces with one instruction space, a graph can be made reentrant. The advantage of this scheme is that reentrancy is accomplished without the overhead of code copying or tagged tokens. Furthermore, since there are fixed memory locations for data token storage, tags are unnecessary. Consequently the tag matching program has been eliminated.

Typically, a dataflow program consists of a collection of separate dataflow graphs (e.g., a separate graph is constructed for each procedure). Each instance of the evaluation of a dataflow graph requires a new data space. In order to simplify the management of memory resources, we restrict the size of an instruction or data space to the size of a memory page (e.g., 1 kbyte). A separate "process" is invoked to evaluate each {instruction space, data space} pair. We call these processes *contexts* to emphasize their small size. (We reserve the term *process* for the execution of a family of related contexts. I.e., the term *process* carries its usual meaning.)

2. Contexts

A context is a small to medium granularity process. A context evaluates a static dataflow graph. The execution of a program requires the dynamic creation and execution of many contexts. In order to perform useful computation, these contexts must communicate. We have adopted a simple communications model based on the use of unidirectional communication channels.

A context requires (1) a static dataflow graph, (2) a data token space, and (3) a pair of communication channels called the *input* and *output* channels of the context. A context receives data tokens over the input channel, evaluates its dataflow graph using the data token space for token storage, and sends its results over the output channel.

In dynamic dataflow machines, conditional execution, iteration, and function invocation are accomplished using special actors that alter the tags associated with tokens. In effect these actors change the "context" in which the tokens are executing. By analogy, in semi-static dataflow, conditional execution, iteration and function invocation are all accomplished by the dynamic creation of contexts.

The essence of the semi-static dataflow approach is the partitioning of the program into a collection of dataflow graphs. Each of these graphs can be evaluated using the static dataflow execution model. That is, data token storage addresses are statically allocated for each of these dataflow graphs. By associating several virtual data spaces with a single graph, the dataflow graph can be made reentrant. These associations are established dynamically as the program is being executed. Each of these associations is executed by a low-level process called a context. Special actors are used for the creation of contexts and for the communication of data tokens between contexts. Thus the semi-static dataflow approach is a cross between the static and dynamic dataflow execution models

3. Communication Primitives

Communication between contexts is accomplished over communication channels. A channel is an abstract entity whose purpose is to provide a unidirectional communications path between two contexts. Communication channels can be established in two ways:

- Communication channels can be opened dynamically during program execution. These channels are created when contexts are created to provide the necessary communication paths between parent and child contexts.
- Communication channels can be allocated statically at compile time. These are channels explicitly declared and used by the programmer.

A channel is used to transfer a fixed length message from one context to another. Each message can carry exactly one (scalar) data token. Consequently, the length of the data portion of a message is equal to the word size of the machine.

Channels use an unbuffered, synchronous communications strategy sometimes called a rendezvous. Intercontext communication is accomplished using two simple primitives called *send* and *receive*. Each channel has a unique channel identifier. This identifier is used by channel primitives to select the desired communications channel. This is essentially the mechanism used in the Occam language for interprocess communication[8], and will be discussed later in relation to our simulation study.

The dataflow actors used to represent the two communication primitives are shown in Fig. 1. The send actor is a three-input, one-output dataflow actor. The first operand, k, is a control token. It is used to sequence communications primitives. Its use will be described below. The second operand, c, is a channel identifier. It specifies the channel to use for the communication. The third operand is the value, x, to be transmitted on the channel. The output of the send

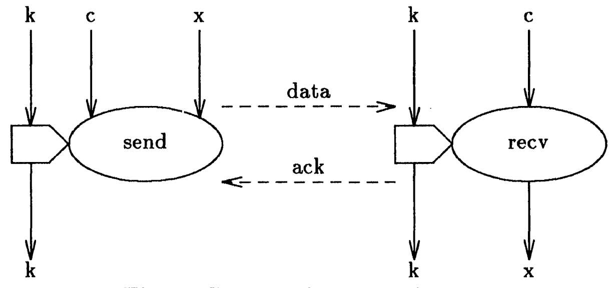


Fig. 1. Communication primitive dataflow actors.

actor, k, is a copy of the input control token.

The receive actor is a two-input, two-output dataflow actor. The first operand, k, is a control token. The second operand, c, is a channel identifier. The first output of the receive actor, k, is a copy of the input control token. The second output of the receive actor is the value, x, received on the channel.

The send and receive actors are non-standard dataflow actors in two senses:

- They are not free from side effects. Both send and receive affect the execution of other contexts.
- They are not strictly functional in the mathematical sense. That is, the result of their execution is not strictly a function of their operands.

Consequently, the order in which send and receive actors are executed will affect the outcome of the computation. To ensure deterministic results, these actors use control tokens. The sole purpose of these tokens is to sequence actors having side effects. The send and receive actors only produce control tokens on their output arcs after successful communication has occurred. This involves the exchange of two messages between contexts. The first message carries the input data token from the send actor to the receive actor. The second message acknowledges receipt of the data token.

The effect of the execution of a {send, receive} pair is the establishment of dynamic arcs between the two actors. These arcs must be established dynamically since, in general, it is not possible to predict which two send and receive actors in a given program will communicate. This uncertainty can arise in two ways. First, the channel identifier used by a primitive need not be statically determined. Second, the invocation of a particular instance of a communication primitive may be conditionally determined. The net effect of the dynamic establishment of arcs is that separate dataflow graphs are "spliced" together at execution time. For this reason, we call this technique dynamic dataflow graph splicing.

4. Function Invocation by Dynamic Dataflow Graph Splicing

In this section, we describe a function invocation paradigm based on the method of dynamic dataflow graph splicing. The function invocation paradigm consists of four phases of execution: context generation, parameter passing, concurrent execution, and result passing.

4.1. Context Generation Phase

In the context generation phase, the calling context

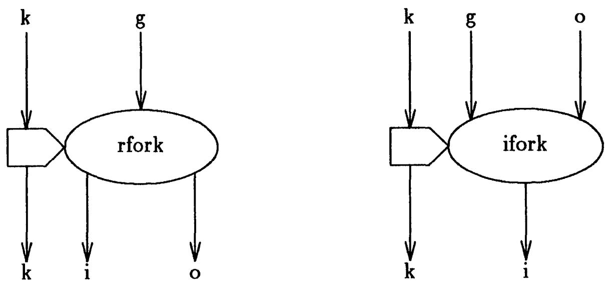


Fig. 2. Context generation primitive dataflow actors.

(i.e., the parent context) creates a new context for the execution of the function. We have currently defined two primitives for the generation of contexts. These primitives are called recursive fork (rfork) and iterative fork (ifork). Function invocation is accomplished using the rfork primitive. (The use of the ifork primitive to implement sequential iteration is described in a subsequent section.)

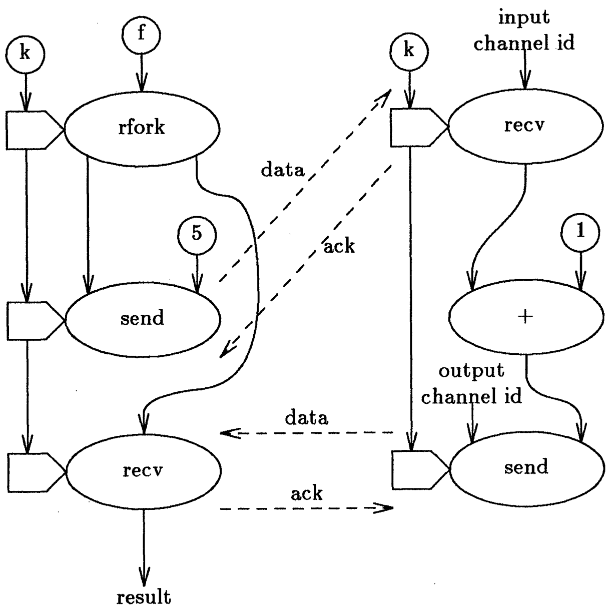
The dataflow actors used to represent the context generation primitives are shown in Fig. 2. The rfork actor is a two-input, three-output dataflow actor. The first operand is a control token, k, which is used to sequence actors with side effects. The second operand is a pointer to the dataflow graph to be evaluated by the child context (i.e., the entry point of the subroutine). The effect of the rfork actor is to generate a new context together with two new channels. These channels become the input and output channels of the child context. The first output of the rfork actor is a control token, k, which is a copy of the input control token. The second and third outputs of the rfork actor, i and o, are the channel identifiers of the input and output channels of the child context. The input channel is used by the parent context to send data tokens to the child context. The output channel is used by the child context to send data tokens to the parent context.

The ifork actor is a three-input, two-output dataflow actor. The first operand is a control token, k, which is used to sequence actors with side effects. The second operand is a pointer to the dataflow graph to be evaluated by the child context. The third operand is a channel identifier. The effect of the ifork actor is to generate a new context together with one new channel. This channel becomes the input channel of the child context. The output channel of the child context is specified by the third operand of the ifork actor. The first output of the ifork actor is a control token, k, which is a copy of the input control token. The second output of the ifork actor, i, is the channel identifier of the input channel of the child context.

4.2. Parameter Passing Phase

In the parameter passing phase, the parent context sends the parameter values to the child context. The parent context does this by sending data tokens over the child context's input channel. The channel identifier for this channel is obtained as a result of the rfork or ifork primitive.

In the event that the child context requires more than one parameter, those parameters must be transmitted to the child sequentially. Consequently, the parent and child contexts must use a prearranged parameter sequence to ensure correctness. This sequence can be specified statically at compile time.



```
proc f (var y, value x) =
  y := x + 1;
var result;
f (result, 5)
```

Fig. 3. Basic function invocation paradigm.

4.3. Concurrent Execution Phase

After the child context has received all its parameter values from the parent context, the concurrent execution phase begins. During the concurrent execution phase, both the parent and child contexts may execute in parallel.

4.4. Result Passing Phase

During the result passing phase, the child context sends its computed results back to the parent context. The child context does this by sending data tokens over its output channel. The parent context can receive the result data tokens using the channel identifier returned by the rfork actor or specified to the ifork actor.

In the event that the child context produces more than one result, those results are returned in a predetermined sequence just as in the parameter passing phase.

4.5. Example

The dataflow graphs of Fig. 3 illustrate the basic function invocation paradigm. This example consists of a trivial function, namely $f(x) \leftarrow x+1$, and a main program that evaluates $f(5)$. The main program has three actors: (1) The rfork actor creates a new context to execute the graph associated with the function f. (2) The send actor transmits the argument, 5, to the new context. (3) the receive actor receives the result from the called context. The graph corresponding to the function f has three actors: (1) The receive actor receives the arguments to f on the input channel. (2) The + actor adds 1 to the argument. (3) The send actor transmits the result of the function to the parent context via the output channel. The dashed arcs in Fig. 3 correspond to the dynamically created arcs that splice the two dataflow graphs together.

5. Sequential Iteration

In this section we describe an iteration paradigm based on the method of dynamic dataflow graph splicing. The essential feature of this paradigm is the creation of a separate context for each iteration. This is done to allow for dynamic loop unravelling, i.e., to allow the parallel evaluation of multiple iterations.

The program of Fig. 4 illustrates the iteration paradigm. The iteration paradigm requires three separate dataflow graphs that are spliced together at execution time. These graphs are: the main program graph (m), the loop body graph (b), and the loop terminator graph (t).

5.1. Main Program Graph

The main program graph uses the basic function invocation paradigm described in the preceding section to initiate the execution of a loop. In effect, the main program graph calls the loop body graph as if it is a function. The main program uses the rfork primitive to create the new context. The main program graph then transmits to the loop body the value of the loop counter and any values used in the body of the loop. In the example of Fig. 4, the main program graph invokes the loop body and transmits the initial values of the variables *sum* and *i*. A consequence of using the basic function invocation paradigm is that the main program graph may execute concurrently with one or more loop iterations.

5.2. Loop Body Graph

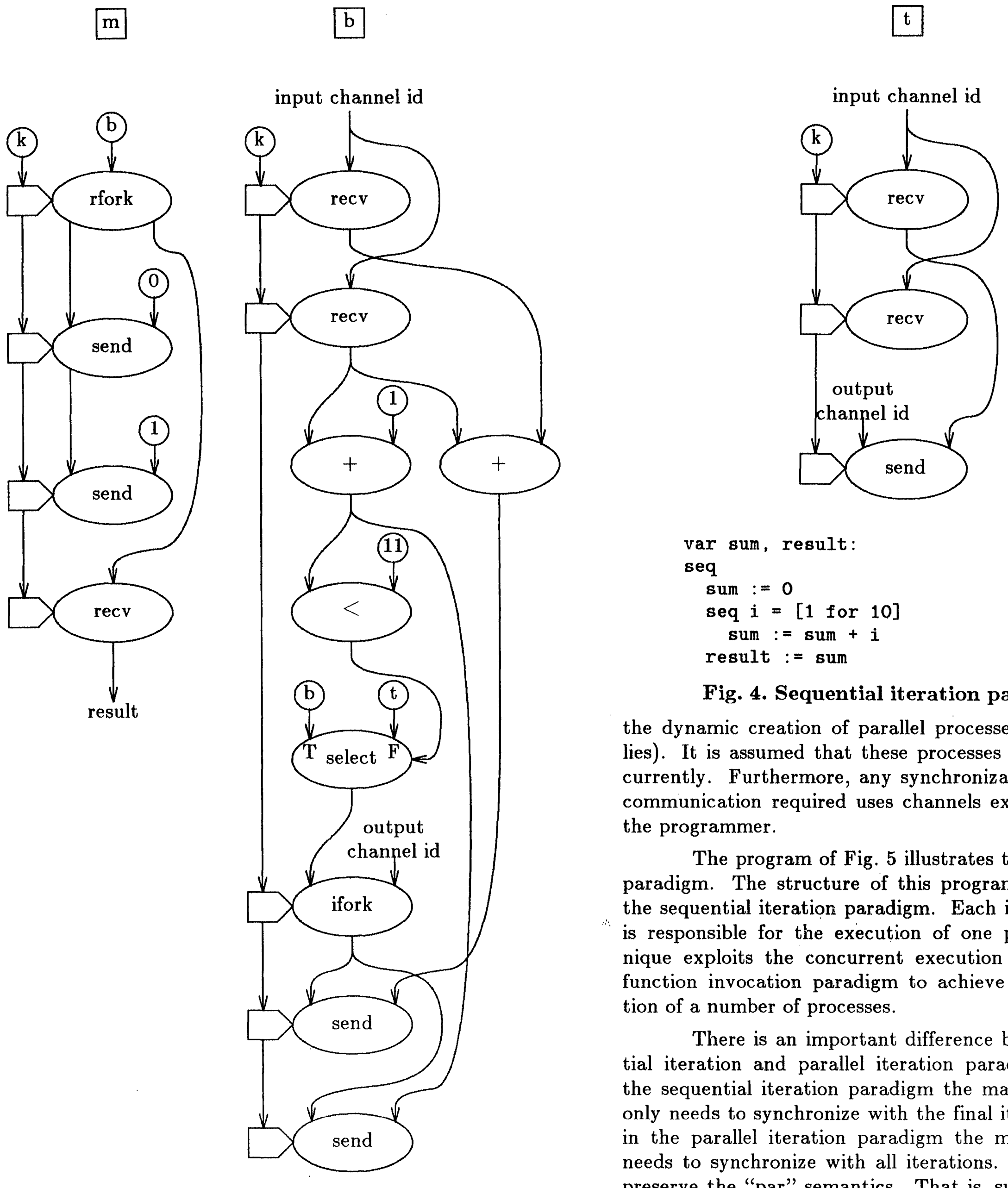
The loop body graph performs the calculations involved in a single iteration of the loop. It then tests the termination condition and creates a new context. This new context will execute the next iteration of the loop if the termination condition is not satisfied. Otherwise, it executes the loop terminator graph.

The loop body graph uses the ifork primitive to create the new context. Recall that the ifork primitive passes on the specified output channel to the child context. This output channel is the channel over which the main program graph expects to receive its results. In this way, the final iteration of the loop can return its results directly to the main program graph without having to pass through all the intermediate loop iterations. This is similar in effect to tail recursion. The advantage of this approach is that it allows loop unravelling yet at the same time permits the recovery of resources allocated to earlier iterations as they terminate.

In the example of Fig. 4, the loop body receives the values of *sum* and *i*. It computes new values for *sum* and *i* and tests the loop termination condition. If the loop termination condition is not satisfied, it invokes a new instantiation of the loop body graph. If the loop termination condition is satisfied, it invokes the loop terminator. In both cases, it transmits the new values of *sum* and *i* to the child context.

5.3. Loop Terminator Graph

The loop terminator graph is used to return the results from the final iteration of the loop back to the main program graph. The loop terminator graph is invoked by the loop body graph when the loop termination condition has been satisfied. Note that the loop body graph and loop terminator graph must have the same call format. That is, the



```
var sum, result:
seq
  sum := 0
  seq i = [1 for 10]
    sum := sum + i
  result := sum
```

Fig. 4. Sequential iteration paradigm.

sequences in which the arguments are transmitted must be identical. However, the loop terminator graph will discard most of the input values except for the final results. These results are transmitted via the output channel. Since the output channel has been inherited from preceding loop iterations, the values transmitted on this channel are returned directly to the main program graph.

In the example of Fig. 4, the loop terminator receives the values of *sum* and *i*. It discards the value of *i* and returns the *sum* to the main program graph. The value received by the main program graph thus becomes the result.

6. Dynamic Process Creation by Parallel Iteration

In this section we describe a modification of the sequential iteration paradigm described above which allows the dynamic creation of parallel processes (or context families). It is assumed that these processes are to execute concurrently. Furthermore, any synchronization or interprocess communication required uses channels explicitly declared by the programmer.

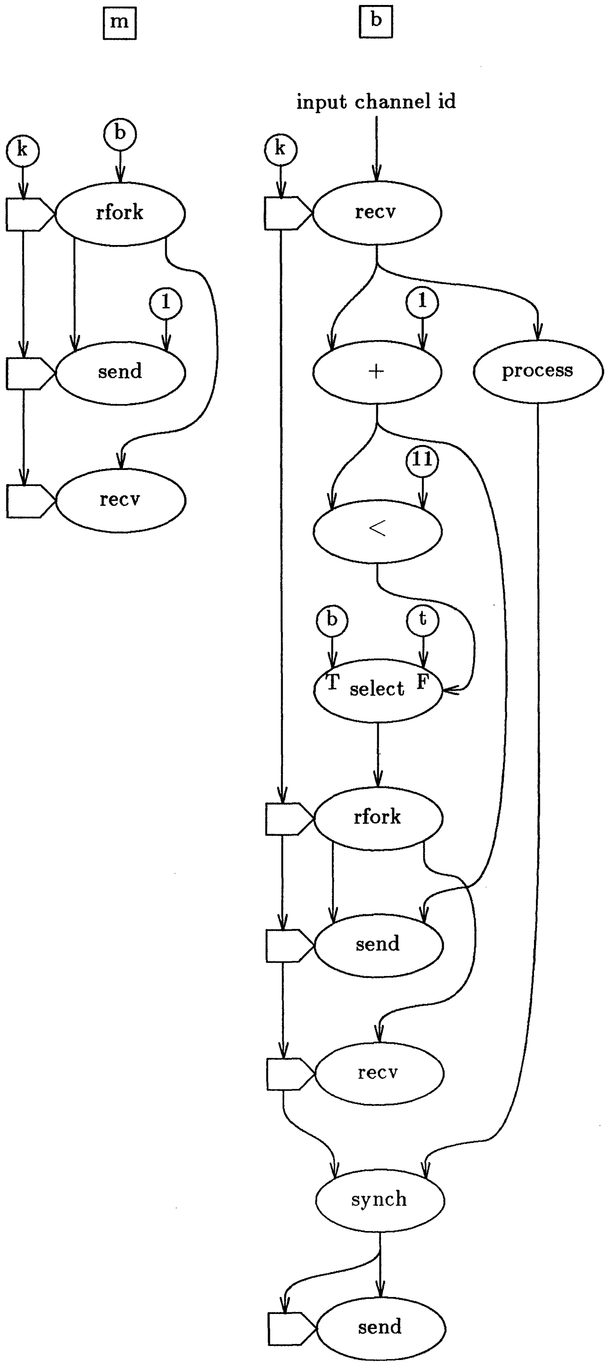
The program of Fig. 5 illustrates the parallel iteration paradigm. The structure of this program is very similar to the sequential iteration paradigm. Each iteration of the loop is responsible for the execution of one process. This technique exploits the concurrent execution phase of the basic function invocation paradigm to achieve the parallel execution of a number of processes.

There is an important difference between the sequential iteration and parallel iteration paradigms. Whereas in the sequential iteration paradigm the main program context only needs to synchronize with the final iteration of the loop, in the parallel iteration paradigm the main program graph needs to synchronize with all iterations. This is required to preserve the "par" semantics. That is, subsequent computation in the main context cannot proceed until all parallel process instantiations have terminated.

Synchronization is explicitly accomplished by message passing. Each iteration *i* waits until its process instantiation has completed and until it has received a message from iteration *i*+1 indicating that all higher number iterations have completed. When both conditions have been met, iteration *i* returns a message to iteration *i*—1 indicating that it has completed. In effect, parallel iteration is analogous to conventional recursion except that all instantiations proceed in parallel.

7. Performance Analysis

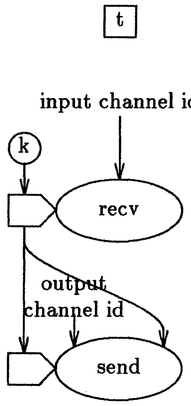
In order to characterize the potential performance of the semi-static dataflow execution model, we have designed a



shared-bus multiprocessor architecture called a *Queue Machine Multiprocessor* which uses the semi-static execution model[9]. In addition, we have implemented a preliminary run-time environment that supports the execution of programs compiled using a prototype Occam compiler. In this section we will present the results obtained by simulation of the execution of a number of benchmark programs.

7.1. Architectural Model

The Queue Machine Multiprocessor is a shared-bus multiprocessor which consists of a number of processing elements each with their own local memory connected by a



```
par i = [1 for 10]
  Process (i)
```

Fig. 5. Parallel iteration paradigm.

shared interprocessor communication bus. The processing elements in this system are called *Queue Machines*[10]. A queue machine is a processing element that has been optimized for the efficient evaluation of static, acyclic dataflow graphs[11]. In contrast to typical dataflow processing elements, a queue machine has a single locus of control. However, we have shown that a queue machine can efficiently utilize a pipelined ALU to exploit the parallelism in a static, acyclic dataflow graph (intracontext parallelism).

We have also implemented a *Queue Machine Multiprocessing Kernel* for the Queue Machine Multiprocessor[9]. This is a collection of software routines in which queue machine programs generated by the Occam language compiler described below can be executed. The purpose of the kernel is to manage contexts (i.e., small processes) and resources (memory and channels). In particular, this kernel provides the context generation primitives rfork and ifork.

7.2. Occam Compiler

The benchmark programs used in this performance analysis are written in the Occam programming language[8]. We have constructed a prototype compiler that translates Occam source programs into dataflow graphs for execution on the Queue Machine Multiprocessor.

The compiler uses the basic function invocation, sequential iteration, and parallel iteration paradigms described above. It automatically partitions the source program into multiple contexts for parallel execution. The current partitioning algorithm is very simple (and, consequently, suboptimal). The compiler generates a separate dataflow graph for every procedure, for every outcome of a conditional branch, and for the body of every loop (sequential and parallel iteration). (The compiler also constructs loop terminator graphs as needed.)

The Occam compiler automatically emits the context generation and intercontext communication primitives as required by the function invocation, and sequential and parallel iteration paradigms. In order to emit the intercontext communication primitives, the compiler must choose a sequence of the input arcs to each dataflow graph. The compiler automatically chooses that sequence which maximizes the amount of computation possible within a context before another input is required. This heuristic has been found, experimentally, to produce good overall throughput results on the multiprocessor[9].

7.3. Benchmark Programs

We have simulated the execution of a number of benchmark programs on the Queue Machine Multiprocessor. In this paper we present the results for four different Occam programs. The tasks performed by the benchmark programs are: 1) Matrix Multiplication, 2) Fast Fourier Transform, 3) Cholesky decomposition, and 4) Congruence transformation.

7.3.1. Matrix Multiplication

The Matrix Multiplication program computes the product of two $N \times N$ matrices.

7.3.2. Fast Fourier Transform

The Fast Fourier Transform program computes the discrete Fourier transform of N data points, where N is a power of 2. The program is based on the binary recursive FFT algorithm described in [12].

7.3.3. Cholesky Decomposition

The Cholesky Decomposition program is an algorithm for the factoring of a symmetric positive definite matrix A of order N into the product LL^T of a lower triangular matrix and its transpose. This algorithm is a transliteration of the dataflow-based algorithm presented in [13].

7.3.4. Congruence Transformation

The Congruence Transformation program computes the congruence transformation of a matrix A. That is, given two $N \times N$ matrices A and Q, it computes the matrix $C = QAQ^T$. This algorithm is based on an algorithm presented in [13].

7.4. Simulation Results

The principle performance metric used in our study is the system throughput ratio. The system throughput ratio, R_n, is defined as the ratio of the throughput on a system with n processors to the throughput on a system with 1 processor. Let X be the total work to be done in some particular workload class. The total workload consists of the sum of the user workload X^U and the kernel workload X^K. Consequently, the total execution time, T_n, is the sum of the user execution time, T_n^U, and the kernel execution time, T_n^K. The throughput of the system is the ratio of the user workload to the total execution time, X^U/T_n. (The contribution to the workload due to the kernel is overhead and is excluded from the throughput calculation.) Thus, $R_n = \dfrac{(X^U/T_n)}{(X^U/T_1)} = \dfrac{T_1}{T_n}$.

The system throughput ratios obtained by simulation of the four benchmark programs described above are shown in Fig. 6. A number of Queue Machine Multiprocessor systems have been simulated having one to eight processing elements. Note that for small values of n (the number of processors), the system throughput ratio is greater than n. In effect, the system exhibits super-linear speedup.

These simulation results can be explained by using a modified derivation of Amdahl's law [14] that takes into consideration the effect of the kernel workload as well as the

Table I
Benchmark program statistics

program	S	G	A
Matrix Multiplication	15	8	236
Fast Fourier Transform	63	17	587
Cholesky Decomposition	74	22	428
Congruence Transformation	66	18	434
S - number of Occam source lines G - number of dataflow graphs A - total number of actors			

user workload. We assume that the user workload consists of two parts: (1) a sequential part that requires a fixed amount of execution time, S^U, and (2) a parallel part, the execution time of which is inversely proportional to the number of processors, P^U/n. Thus, $T_n^U = \dfrac{P^U}{n} + S^U$. The kernel workload consists of a large number of calls to kernel primitives. We assume that these calls can execute independently. Therefore, $T_n^K = Y/n$, where Y is proportional to the total work done by all the kernel calls. It is important to observe that some of the kernel workload is proportional to the number of processes running on a given processor. Thus, it is argued that $Y = \dfrac{P^K}{n} + S^K$. Hence, $R_n = \dfrac{1}{1+(1/n-1)f+(1/n^2-1)g}$, where $f = \dfrac{S^K+P^U}{P^K+S^K+P^U+S^U}$, and $g = \dfrac{P^K}{P^K+S^K+P^U+S^U}$.

Note that $\lim_{n\to\infty} R_n = \dfrac{1}{1-(f+g)}$. That is, the system throughput ratio has a finite upper bound. Also $\left.\dfrac{dR_n}{dn}\right|_{n=1} = f+2g$. That is the slope of the graph of R_n vs. n is $f+2g$ at $n=1$. Note that when $S^U < P^K$, $f+2g > 1$. That is, for small values of n, the slope of the graph of R_n vs. n can be greater than 1. This situation is called super-linear speedup.

Although it may seem unlikely, this kind of super-linearity is in fact a real phenomenon. Super-linear results have been reported in [15]. One way to view super-linearity is that is the manifestation of a kernel implementation that penalizes systems with a small number of processing elements and that becomes more efficient only when the number of processing elements is increased.

Note that if we neglect kernel overhead, $f = \dfrac{P^U}{P^U+S^U}$ and $g=0$. In this case, the system throughput ratio can be simplified to $R_n = \dfrac{1}{1+(1/n-1)f}$ (Amdahl's law)[14].

8. Conclusions

In this paper we have presented a new dataflow execution model called semi-static dataflow. Programs are partitioned into a collection of dataflow graphs. Each of these graphs is evaluated by a low level process (called a context) using the static dataflow execution model. Contexts are dynamically created during execution to implement iteration, conditional execution, and function invocation. This execu-

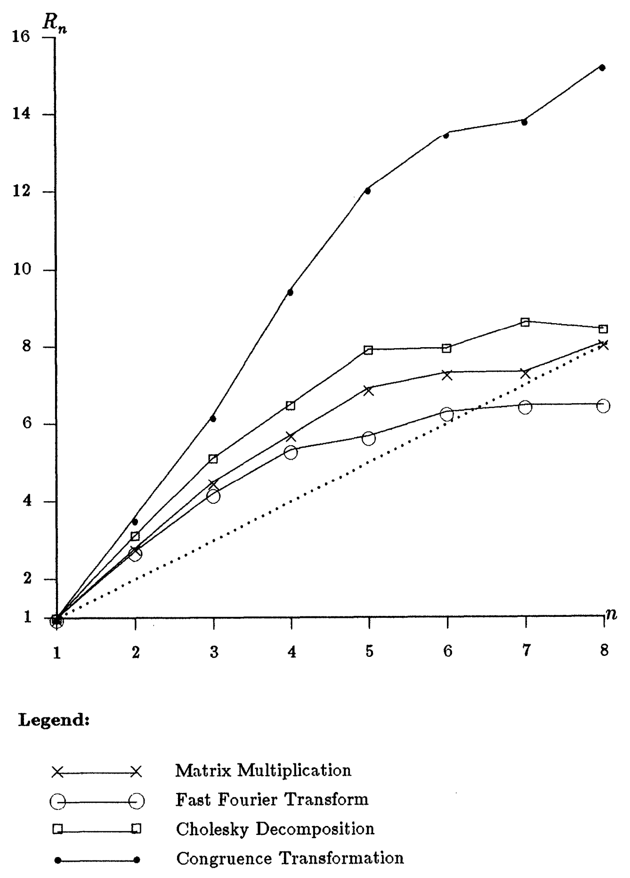


Fig. 6. System throughput ratios for the benchmark programs vs. number of processors.

tion model provides the benefits of both static and dynamic dataflow architectures without the associated costs of code copying or tag manipulation.

We have also presented a number of program structure paradigms for function invocation, sequential iteration, and parallel iteration. These paradigms are used by a prototype compiler for the Occam programming language to generate semi-static dataflow program graphs. This compiler has been used to generate code for a semi-static dataflow multiprocessor. We have presented the simulation results for several benchmark programs. These results show that the semi-static execution paradigm is capable of automatically exploiting the increased parallelism available as the number of processors in the multiprocessor system is increased. We have used a modified derivation of Amdahl's law to justify the simulation results.

9. References

1. V. P. Srini, "An Architectural Comparison of Dataflow Systems," *Computer*, Vol. 19, No. 3, pp. 68-88, Mar. 1986.
2. K. W. Todd, "Function Sharing in a Static Data Flow Machine," *Proc. 1982 Int. Conf. Parallel Processing*, pp. 137-139, IEEE, Aug. 1982.
3. J. B. Dennis, "Data Flow Supercomputers," *Computer*, Vol. 13, No. 11, pp. 48-56, Nov. 1980.
4. A. L. Davis and R. M. Keller, "Data Flow Program Graphs," *Computer*, Vol. 15, No. 2, pp. 26-41, Feb. 1982.
5. G. S. Miranker, "Implementation of Procedures on a Class of Data Flow Processors," *Proc. 1977 Int. Conf. Parallel Processing*, pp. 77-86, IEEE, Aug. 1976.
6. Arvind and K. P. Gostelow, "The U-Interpreter," *Computer*, Vol. 15, No. 2, pp. 42-49, Feb. 1982.
7. I. Watson and J. Gurd, "A Prototype Data Flow Computer with Token Labelling," *1979 Nat. Comp. Conf.*, Vol. 48, pp. 623-628, June 1979.
8. INMOS Limited, *OCCAM Programming Manual*, Englewood Cliffs, NJ: Prentice-Hall, 1984.
9. B. R. Preiss, "Data Flow on a Queue Machine," Ph.D. Thesis, Univ. of Toronto, Dept. of Elec. Engin., Toronto, Ontario, Canada, 1984.
10. B. R. Preiss and V. C. Hamacher, "Data Flow on a Queue Machine," *Conf. Proc. 12th Ann. Symp. Comp. Arch.*, pp. 342-351, IEEE Computer Society Press, June 1985.
11. B. R. Preiss, "Design and Simulation of a Data-Flow Multiprocessor System," M.A.Sc. Thesis, Univ. of Toronto, Dept. of Elec. Engin., Toronto, Ontario, Canada, 1984.
12. J. D. Lipson, *Elements of Algebra and Algebraic Computing*, Reading, MA: Addison-Wesley, 1981.
13. D. P. O'Leary and G. W. Stewart, "Data-Flow Algorithms for Parallel Matrix Computations," *Communications of the ACM*, Vol. 28, No. 8, pp. 840-853, Aug. 1985.
14. J. P. Riganati and P. B. Schneck, "Supercomputing," *Computer*, Vol. 17, No. 10, pp. 97-113, Oct. 1984.
15. J. Sanguinetti, "Performance of a Message-Based Multiprocessor," *Computer*, Vol. 19, No. 9, pp. 47-55, Sept. 1986.

Using Control States for Parallelism Extraction*

Tom Bennet
Computer Science Department
University of Maryland, College Park
College Park, MD, 20742, USA

Abstract – Existing work on parallelism extraction usually occupies one of two categories: DO-loop methods which use the loop index value space for analysis, and stream methods which examine dataflow graphs. Our method combines some of the advantages of the loop and stream approaches through a generalization of the DO loop index variable called a control state. With control states, we can deal with **while** loops and with loop bodies containing **if**'s. We describe the control state concept and how it can be used to extract parallelism.

Introduction

The advantages of parallel computers are generally recognized, but many questions remain about how best to program them. Special programming languages have been proposed, some of which allow explicit expression of parallel operations [13] [4], and some which provide less-sequential forms of familiar operations [1] [8]. Still, it would also be nice to be able to compile programs written in standard sequential programming languages into code to run on parallel machines.

Existing work on parallelism extraction usually falls into one of two categories: DO-loop analysis (e.g. [3], [12], [9], and [2]), which uses the loop index values to analyze the whole execution of a loop at compile time, and stream analysis (e.g. [7] and [5]), which examines dataflow graphs. Much of the DO-loop work is quite elegant and precise because it benefits from the space of loop index values as a conceptual underpinning, but the domain is rather limited: **while** loops and **if**'s in loop bodes are particular problems. The stream analysis work is much more general, but it lacks the nice conceptual framework, making it much more ad hoc.

Our method combines some of the advantages of the loop and stream approaches through the concept of a control state. A control state is a generalization of the DO loop index variable that applies to **while** loops and deals better with **if**'s inside loops. We describe the control state concept and, through an example, how it can be used to extract parallelism.

Control States

Description

We view a computation as a sequence of states, each state after the first generated from the previous one. A state specifies the current values of all the variables and the operation applied to it to produce the next state. For each state, there is a set of input variables and a set of output variables. (These sets do not depend on the operation alone which may use array subscripting or pointer dereferencing.) As such, the computation forms a graph where the nodes are states and there is an arc from state i to state j, $i < j$, if and only if state i outputs some variable which j reads, and no intervening state outputs that variable.

If the nodes of this graph were assigned to different processors in a parallel machine, the information each processor would need, in addition to the input values, constitutes the "control state." Specifically, a control state is any subset of a state large enough to:

- Uniquely identify the state within the computation,
- Determine the set of input and output variables for the identified state, and
- Determine which states read the output variables of the identified state.

Note that this is enough information to determine the graph.

For example, consider the following code segment that might be part of an output conversion program:

```
s1  num := M;
s2  sub := N;
s3  while num != 0 do
s4     dig := num mod 10;
s5     num := num div 10;
s6     arr[sub] := chr(ord('0')+dig);
s7     sub := sub-1
s8  od;
s9  print(arr,sub+1)
```

The computation for the program when **M** = 23 and **N** = 4 is given in Figure 1, and the input and output sets for each state are given in Figure 2. The graph formed is given in Figure 3. There is a state for every execution of every non-control statement. Control statements are omitted since they do not produce values, but their meaning is incorporated since each state must determine the next state. The variable **@**, referred to as the pc, designates the state's next operation.

A series of control states for each state in this computation is given in Figure 4. Note that the control states satisfy the above conditions:

- No control state is a subset of any state other than the one from which it was taken. Therefore each control state identifies its state.
- The pc alone determines the inputs for all states and the output

state	@	num	sub	dig	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
R_1	s1	?	?	?	?	?	?	?	?
R_2	s2	23	?	?	?	?	?	?	?
R_3	s4	23	4	?	?	?	?	?	?
R_4	s5	23	4	3	?	?	?	?	?
R_5	s6	2	4	3	?	?	?	?	?
R_6	s7	2	4	3	?	?	?	?	'3'
R_7	s4	2	3	3	?	?	?	?	'3'
R_8	s5	2	3	2	?	?	?	?	'3'
R_9	s6	0	3	2	?	?	?	?	'3'
R_{10}	s7	0	3	2	?	?	?	'2'	'3'
R_{11}	s9	0	2	2	?	?	?	'2'	'3'
R_{12}	?	0	2	2	?	?	?	'2'	'3'

Figure 1: A Computation for the Conversion Program.

state	in	out
R_1	M,@	num,@
R_2	N,@	sub,@
R_3	num,@	dig,@
R_4	num,@	num,@
R_5	sub, dig,@	arr[4],@
R_6	sub,@	sub,@
R_7	num,@	dig,@
R_8	num,@	num,@
R_9	sub, dig,@	arr[3],@
R_{10}	sub,@	sub,@
R_{11}	arr[4], arr[3], sub,@	@
R_{12}	-	-

Figure 2: The input and output sets for each state.

* This work was sponsored by the United States Air Force under grant number AFOSR 87-130.

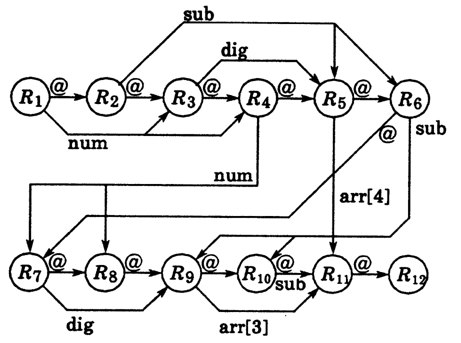


Figure 3: Graph of Example Conversion Computation.

control state	@	num	sub
A_1	s1	?	?
A_2	s2	23	?
A_3	s4	23	4
A_4	s5	23	4
A_5	s6	2	4
A_6	s7	2	4
A_7	s4	2	3
A_8	s5	2	3
A_9	s6	0	3
A_{10}	s7	0	3
A_{11}	s9	0	2
A_{12}	?	0	2

Figure 4: Control states for conversion program computation.

variable for all states except R_5 and R_9. In those, the output variable **arr[sub]** is determined by **@** and **sub**, which are both included in the control states.

- Outputs from states R_5 and R_9 are always used in the next state after the loop exits (i.e., R_{11}), and no variables are needed to determine that. Other values are used either in specific states in the same iteration, again requiring no variables to compute, or are used in specific states in the next iteration, if any. The "if any" is easily computed from **num** in the control state, i.e., for states with **@** values s4 and s5, see if $|\text{num}| \geq 10$; for s6 and s7, see if **num** != 0.

Note also that if we left **sub** out of all the control states except A_5, A_9, and A_{11}, they would still be control states. We could also leave out any variable which is undefined, that is, has a value of "?".

The set of variables in a control state is called a control variable set, or *CVS*. If all control states for a computation have the same *CVS*, it is the *CVS* for the whole computation.

Relationship to Previous Work

The control state is similar to the combination of the **DO**-loop index variables and the **@** variable. This equivalence has been noticed before: [6] suggests that in a loop body with more than one statement, the statement name could be treated as an extra inner loop index. Conversely, a **DO** loop nest with a one-statement body produces computations for which the loop index set alone is a *CVS*.

In the conversion example, **num** is analogous to the loop index, and **sub** is included because it helps determine an output variable set. Since existing loop analysis methods generally concern themselves with loops where the subscript expressions are functions of the loop index variables [11], the pair of **num** and **sub** is analogous to the loop index variable.

Existing **DO** loop methods break dependences by introducing new variables, e.g., Renaming and Scalar Expansion [10]. Control states do this naturally because, while separate values for some variable **x** may be produced in any number of (control) states, each is treated separately since the value is identified not just by the variable name **x**, but also its graph position.

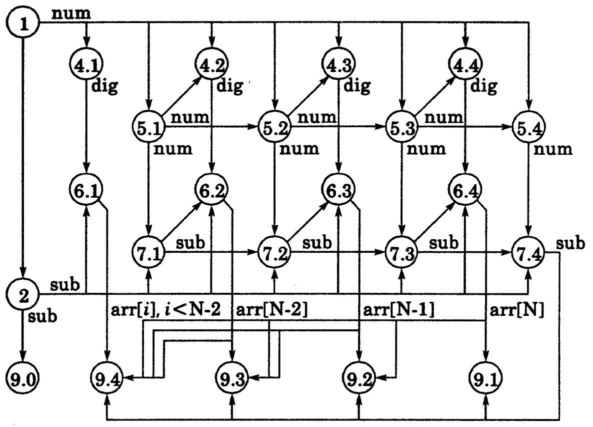


Figure 5: Static Graph for Conversion Program.

Direct Application

This section describes how control states can be used to extract parallelism. We define a finite "static graph" which can represent the set of all computations of some program, and the execution cost in terms of it. We then discuss the problem of allocating the graph nodes to processors in order to get a good execution time.

Node Completion times

We model execution time by assigning each node a completion time defined in terms of its parents', accounting for communication costs and also its own execution time. Each node begins sending data at its completion time (*ct*). Each datum is sent serially requiring the transmit time (*xt*), traverses the communications medium for its propagation delay time (*pd*), and then is available to be read at the receiver, which must read its inputs serially, each taking its receive time (*rt*). After all data are received, a node computes its function consuming its execution time (*et*). The model is designed to be general enough to permit the "communications medium" to be either a message-passing network of some kind or a global memory.

The Static Graph

In order to manipulate programs rather than processes, we define a "static graph" representing all computations that can be executions of some given program. Each node in the static graph represents some set of states from some represented computation(s). For each arc $\langle a, b\rangle$, there is a "continuance probability," *cp*, which has the meaning "when a computation enters a state S represented by node a, the probability that it will next enter a state R represented by b is given by $cp(\langle a, b\rangle)$."

There is an arc from node a to node b in the static graph whenever a represents some state S and b some state R such that $\langle S, R\rangle$ is an arc in the graph of some computation, except that:

- Each static node must receive enough information to constitute a control state for any represented state, so it may need to receive data that is not in the input set of any represented state.
- If a received value is the same for all represented states, there is no need to receive it at all.

Figure 5 contains a static graph for the conversion program. In it, each node represents sets of states having the same pc value, so that it need not be transmitted. The nodes are labeled with the statement number of the **@** value for their represented states, and a second figure when needed to give unique names. The nodes 1 and 2 represent states in which the pc is statement s1 and s2. Nodes 4.4, 5.4, 6.4,

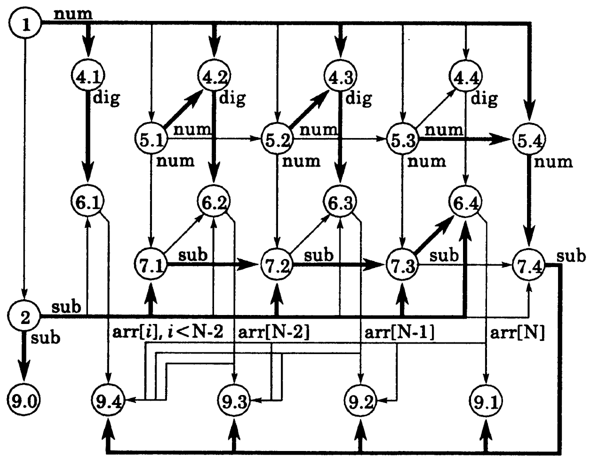


Figure 6: Control Trace for Conversion.

and 7.4 represent states in which the pc is statement s4, s5, s6, or s7, respectively, and which are part of the last execution of the loop body. Nodes 4.3 – 7.3 likewise represent those statements for next-to-last iterations, etc. The 4.1 – 7.1 group represent all the previous iterations. Nodes 9.0 – 9.4 represent states where the pc is s9 for different numbers of digits generated (or numbers of iterations completed). Node 1 is a parent of all of the s4 and s5 nodes because they all represent some states which are part of the first loop iteration and hence use the initial values. Node 2 is likewise a parent of all s6 and s7 nodes. The completion time of a static graph node is an average of the completion times of the states it represents, with one additional cost: the sink computation time *sc*. That is the time required to compute where to send the output values. It is spent by the sender just before the *xt* begins.

The continuance probabilities of $\langle 4.x, 6.x\rangle$ and $\langle 5.x, 7.x\rangle$ are all 1 because the whole loop body is executed whenever it is entered. Arcs $\langle 5.x, 4.x+1\rangle$ and $\langle 7.x, 6.x+1\rangle$ also have probabilities of 1 because the sources of those arcs represent only states which are not part of the last iteration of the loop. Other probabilities depend on the actual value of `M`, but, assuming that the probability of the loop test being true is a constant p, $cp(\langle 2, 9.0\rangle) = 1-p$, $cp(\langle 1, 4.4\rangle) = p(1-p)$, $cp(\langle 1, 4.3\rangle) = p^2(1-p)$, $cp(\langle 1, 4.2\rangle) = p^3(1-p)$, and $cp(\langle 1, 4.1\rangle) = p^4$. Node 4.1 breaks the pattern because it covers more than a single iteration. Other *cp* values are computed similarly. The model has predicted actual execution times on the ZMOB [14] MIMD processor to within 10%.

Processor Allocation

Processor allocation can be indicated by drawing heavy "control trace" arcs on the graph which indicate ordering of nodes due to execution by the same processor and may replace data dependence arcs. When communicating states are assigned to the same processor, the communication times become zero. We used a heuristic search to find an allocation that provides a good completion time for the graph of Figure 5, and the result is given in Figure 6. In the heuristic solution, each $\langle 5.x, 4.x+1, 6.x+1\rangle$ is a stage which extracts one digit from `num` as it is passed along. The $\langle 7.1, \ldots\rangle$ series provides the subscript for each stage. In the loop itself (ignoring the print routine which is expensive and atomic), the allocation found 32% of the cost savings that could be realized if the graph were executed with free communications. This works out to about two communications per iteration. The calculations are summarized in Figure 7.

Time for sequential execution	82.5
Time for parallel execution as allocated	71.2
Speedup	82.5/71.2 = 1.16
Time for parallel exec. with free communication (min)	47.2
Speedup	82.5/47.2 = 1.75
Time for parallel exec., 1 node/processor (max)	91.5
Savings found by allocation	82.5 − 71.2 = 11.3
Savings when using free communications	82.5 − 47.2 = 35.3
Percent of max savings found by allocation	11.3/35.3 = 32%
Cost of *xt* or *rt*	8
Savings shortfall in terms of communications cost	(35.3 − 11.3)/8 = 3.0
Expected number of loop iterations	1.5
Savings shortfall, communications per iteration	3.0/1.5 = 2.0

Figure 7: Conversion Loop Allocation Statistics.

Translating programs to static graphs

We need to discover a set of variables which is a *CVS* for every execution of a program, or part of a program, being examined. For sequential code, this is not usually hard. For `while` loops it may be more difficult, but we have an algorithm that can find reasonable control states. It is based based on a theorem which follows from our formal definitions of computation and control state. Before stating the theorem, a few preliminaries:

- A state is represented by a set of ordered pairs of a variable and its value.
- Unless otherwise indicated, A is a state.
- If A is a state and v is a variable, $A(v)$ gives the value of v in state A. Therefore, the notation $A(@)$ gives A's function (statement), $A(@)(A)$ applies it to A and hence gives the next state after A, and $A(@)(A)(@)$ gives the function of the state after A (the next statement).
- For any state A and any set of variables D,
$$A_D \equiv \{\langle v, d\rangle | \langle v, d\rangle \in A \wedge v \in D\}$$
- For any states A_1 and A_2, the set of matching variables is
$$M(A_1, A_2) \equiv \{v | A_1(v) = A_2(v)\}$$
- For any state A, $in(A)$ and $out(A)$ give its input and output variables, respectively.

And the theorem is:

Consider the computation:

$$A_1^1, \ldots, A_n^1, A_1^2, \ldots, A_n^2, \ldots, A_1^k, \ldots, A_n^k, Z \tag{1}$$

Now, if

$$A_p^q(@) \neq A_r^q(@), \qquad 1 \le q \le k, \quad 1 \le p, r \le n, \quad p \neq r \tag{2}$$

and

$$A_p^q(@) \neq Z(@), \qquad 1 \le q \le k, \quad 1 \le p \le n \tag{3}$$

Then, for any set D having the properties:

$$\begin{array}{l} \text{There exists some function } w : P \rightarrow \textit{boolean} \text{ s.t.} \\ \quad w(A^i_{1D}) = T, \qquad 1 \le i \le k \\ \quad w(Z_D) = F \end{array} \tag{4}$$

and

$$out(A_i^j) \cap D \neq \emptyset \Rightarrow in(A_i^j) \subseteq D, \qquad 1 \le j \le k, \quad 1 \le i \le n \tag{5}$$

and

$$\begin{array}{l} D \cup \{@\} \subseteq M(A_i^j, A) \Rightarrow in(A_i^j) = in(A) \wedge out(A_i^j) = out(A) \\ \quad \wedge A_i^j(@)(A_i^j)(@) = A(@)(A)(@) \end{array} \tag{6}$$

the set $C = D \cup \{\mathtt{@}\}$ is a CVS for the computation A^j_1.

Requirements (2) and (3) simply insist that each $A^j_1, \ldots, A^j_n$ series is a loop body execution and that Z is the state after loop exit so that all executions are represented. In (4), the function w represents the while loop test and the condition insists that D contains the variables needed to evaluate it. Condition (5) insists that any change to a value in D must be a function of the D values, and (6) requires that D contain enough variables to determine the input and output sets and the outcome of any `if` tests in the loop body.

The algorithm that comes from this theorem is as follows: Assume a loop of the form **while** e **do** s1;. . .;sn **od**.†

Let

V be the set of all variables,
Q be a set containing all states of all loop computations, and
P be a set containing all states.

Define $IN()$, $OUT()$, and CIN any way such that:

$$IN(x) \supseteq \{v | (\exists A)(A \in Q \wedge v \in V \wedge A(\mathtt{@}) = x \wedge v \in in(A))\}$$

That is, all possible input variables for statement x. For instance, $IN(\mathtt{a} := \mathtt{b}) = \{\mathtt{b}\}$, or, assuming $1 \ldots \mathtt{N}$ are the bounds of `v`, $IN(\mathtt{a} := \mathtt{v[i]}) = \{\mathtt{i}, \mathtt{v[1]}, \ldots, \mathtt{v[N]}\}$. Technically, the expression says $IN(x)$ must include all input variables of all states of the computation A which have $A(\mathtt{@}) = x$.

$$OUT(x) \supseteq \{v | (\exists A)(A \in Q \wedge v \in V \wedge A(\mathtt{@}) = x \wedge v \in out(A))\}$$

That is, all possible output variables for statement x.

$$CIN \supseteq \{v | (\exists A)(A \in P \wedge v \in V \wedge A(\mathtt{@}) = \text{“}\mathtt{b} := \text{e”} \wedge v \in in(A))\}$$
for e of the loop above.

That is, all input variables for the loop control expression, e. The variable `b` is an arbitrary boolean variable used to make an assignment statement.

Then the algorithm

```
D := $CIN$;
repeat
    change := false;
    for each $x \in \{s1, \ldots, sn\}$
        if $(OUT(x) \cap D \neq \emptyset \vee$
                    $(\exists A \in Q)(\exists A' \in P)(D \cup \{@\} \subseteq M(A, A') \wedge$
                        $(in(A) \neq in(A') \vee out(A) \neq out(A') \vee$
                            $A(@)(A)(@) \neq A'(@)(A')(@)))) \wedge$
                $IN(x) \not\subseteq D$ then
            D := D $\cup IN(x)$;
            change := true
        fi
    rof
until not change;
C := D $\cup \{@\}$
```

produces a value of C which satisfies (1). To see that this is so, notice that the computations of the `while` loop must satisfy (2) and (3). The conditions on D, (4), (5), and (6), must be true of D for the algorithm to terminate. Since $D \subseteq IN(\text{s1}) \cup \ldots \cup IN(\text{sn}) \cup CIN$, the **while** loop must eventually halt so long as all $IN(x)$ sets and CIN are finite. This algorithm finds the CVS $\{\mathtt{num}, \mathtt{dig}, \mathtt{sub}, \mathtt{@}\}$ for the conversion loop. Obviously this is not optimal, but it is of reasonable size. Another example is the array reversing loop

† As the theorem is currently formulated, there may be no nested `while` loops, and any `if` statements in the loop body must have equal numbers of statements in each alternative. We expect to remove these restrictions later.

```
i := 1;
while i < N/2 do
   t := a[i];
   a[i] := a[N - i + 1];
   a[N - i + 1] := t;
   i := i + 1
od
```

For this, the algorithm finds that $\{\mathtt{i}, \mathtt{N}, \mathtt{@}\}$ is a CVS. So this algorithm permits us to find some reasonable CVS's, and we expect it will be possible to find better or more specialized algorithms. We have not investigated the question of finding the best CVS, or even what the best CVS is: A larger CVS incurs more communication cost, but may reduce the *sc*.

Translating Static Graphs to Programs

First, we extract a set of "control trees" from the graph. Each node that does not have a control trace entering it is the root of the control tree formed by the outgoing control arc(s) and their successor(s). There are six such trees in Figure 6: T_1 rooted at node 1, T_2 at 2, T_3 at 5.1, T_4 at 5.2, T_5 at 5.3, and T_6 at 4.4. T_2 is shown in Figure 8. When a node has multiple control arcs out, it means that after it executes, at most one of its control children will actually follow, whichever represents the state that actually occurs. Each tree, then, represents the program that will be executed by some process, where the process will follow one path from the root to a leaf and exit. In some cases, termination of the process an alternative, represented by an empty subtree, and so not easily drawn. Next, note that nodes can be considered "execution equivalent" when they differ only in what is received, so they can be represented by identical object code. This lets us compress the trees as shown in Figure 8.

Next, we extract the programs for each unique compressed tree by tracing it out, starting at the root and passing through to the leaves. At each node, we first generate a **receive** operation for each datum required from another trace, then generate the code for the node itself, then generate **send**s for each output datum that must be shipped elsewhere. If the node has multiple successors in the tree, an **if** must be generated to determine which will actually be executed. When a node is its own successor, each execution followed by a successor test constitutes a loop in the object program, as in T_2. The results for the tree in Figure 8 is given in Figure 9.

Computation of sinks and the if tests that result from tree branching in this example are all of the form "does the loop execute at least n more times." We have assumed that the translator can see, from the fact that s5 is the only body statement that changes the loop test result, that the question can be answered for any n and `num` by an expression of the form `num ( div 10 )`^{n} `!= 0`. It is a simple optimization to substitute $|\mathtt{num}| >= 10^n$, though we have not done that. In general, the test is performed by seeing how the loop body modifies the control state.

The three communications operations are **send**, **createsend**, and **receive**. **Send** and **createsend** both transmit a particular variable (name and value) to a named process; **createsend** creates the process before sending to it. **Receive** reads in a variable regardless of who sent it: The control state scheme is designed around the idea that when you send data, you know where it goes. In general, a process can be uniquely named by the control state of any node in its tree; we have used a form of this, but with a smaller name size. We identify some trees as executed at most once by the fact that they always perform some code which is outside of any loop in the original program (T_1, T_2, T_5, and T_6), and name these by the tree type. For the others, each root node has a CVS of $\{\mathtt{num}, \mathtt{@}\}$, and the tree type determines the value of `@`, so these are named by the tree name and `num` value.

Current Directions

We are continuing work toward building a prototype compiler

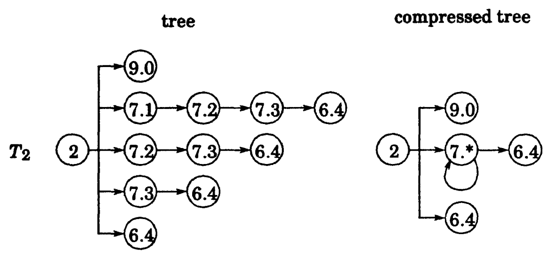


Figure 8: Example execution tree for Figure 6.

```
T₂: receive(num);
2   sub := N;
    if num != 0 then
       if num div 10 != 0 then
          do
             receive(num);
7.*          sub := sub - 1;
             send(< T₃₄ >,sub)
          while num != 0;
          receive(dig);
6.4       arr[sub] := chr(ord('0') + dig);
          send(< T₅ >,arr[sub])
       else
          receive(dig);
6.4       arr[sub] := chr(ord('0') + dig);
          send(< T₁ >,arr[sub])
       fi
    else
9.0    print(arr,sub + 1)
    fi
```

Figure 9: Program derived from the tree, in Figure 8.

based on the control state concept. One area for additional work is dealing with a limited number of processors. Our exposition assumes a run-time environment that manages processors, but another approach is to produce only allocations that use no more than the actual number of processors, as in [15]. This would involve a step after the heuristic allocation that would continue adding control traces so as to chain trees together. This would continue until the number of processes is reduced to the limit.

Another important task will be improving the allocation heuristic. For instance, consider the hexadecimal conversion program:

```
num := M;
sub := N;
while num != 0 do
   dig := num mod 16;
   num := num div 16;
   if dig < 10 then
      arr[sub] := chr(ord('0')+dig)
   else
      arr[sub] := chr(ord('a')+dig-10)
   fi
   sub := sub-1
od;
print(arr,sub+1)
```

Its graph is really no harder to produce than the one in Figure 5, having two nodes in place of each node 6.x, but our studies with the heuristic search have been disappointing. While the occurrence of **if** statements inside loop bodies causes problems for other extraction methods, our formalism has no particular trouble with this, and we feel this is a particular strength of our method which we can exploit as soon as the heuristic is improved.

References

[1] W. B. Ackerman, "Data Flow Languages," *Proc. NCC* (1979), pp. 1087-1095.

[2] R. Allen, D. Callahan, and K. Kennedy, "Automatic Decomposition of Scientific Programs for Parallel Execution," *Proc. Symp. on POPL* (1987), pp. 63-76.

[3] U. Banerjee, *Data Dependence in Ordinary Programs*, Computer Science Dept., University of Illinois at Urbana, UIUCDCS-R-76-837, (November, 1976), 41 pp.

[4] P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE TSE* (June, 1975), pp. 199-207.

[5] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, "Parallel Processing: A Smart Compiler and a Dumb Machine," *Proc. of the SIGPLAN Symp. on Compiler Construction* (1984), pp. 37-47.

[6] J. A. B. Fortes, and D. I. Moldovan, "Parallelism Detection and Transformation Techniques Useful for VLSI Algorithms," *J. of Parallel and Distr. Comp.* (September, 1985), pp. 227-301.

[7] C. C. Foster, and E. M. Riseman, "Percolation of Code to Enhance Parallel Dispatching and Execution," *IEEE Tr. on Comp.* (December 1972), pp. 1411-1415.

[8] D. Gelernter, N. Carriero, S. Chandran, and S. Chang, *Parallel Programming in Linda*, Computer Science Dept., Yale University, YALEU/DCS/RR#359, (January, 1985), 21 pp.

[9] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The Structure for an Advanced Vectorizer for Pipelined Processors," *Fourth Intl. Conf. on Computer Software and Applications* (1980), pp. 709-715.

[10] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Grahps and Compiler Optimizations," *Proc. Symp. on POPL* (1981), pp. 207-218.

[11] D. J. Kuck, "A Survey of Parallel Machine Organization and Programming," *Computing Surveys* (March, 1977), pp. 29-59.

[12] L. Lamport, "The Parallel Execution of DO Loops," *C. ACM* (February, 1974), pp. 83-93.

[13] R. H. Perrott, "A Language for Array and Vector Processors," *ACM TOPLAS* (October, 1979), pp. 177-195.

[14] C. Rieger, *ZMOB: Hardware from a User's Viewpoint*, Computer Science Dept., University of Maryland College Park, TR-1042, (April, 1981), 23 pp.

[15] V. Sarkar, and J. Hennessy, "Compile-time Partitioning and Scheduling of Parallel Programs," *Proc. of the SIGPLAN Symp. on Compiler Construction* (1986), pp. 17-26.

Processor Scheduling Algorithms for Constraint-Satisfaction Search Problems

K. S. Natarajan
Vivek Sarkar

IBM Research
T. J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

Constraint-satisfaction problems arise frequently in Artificial Intelligence and engineering design applications. These problems are computationally intensive and would significantly benefit from speedup through parallel processing. In this paper, we investigate parallelizations of the Forward-Checker algorithm, which is known to be an efficient sequential algorithm for constraint-satisfaction problems. We present two parallel algorithms -- the Threshold Depth-First Priority (TDFP) and the Breadth-First List Scheduling (BFLS) algorithms. Simulation results show that both algorithms are suitable for solving constraint-satisfaction problems in parallel, and yield near-linear speedup even beyond 100 processors. The best choice of algorithm depends on the amount of imbalance in the search problem and the overhead in the target multiprocessor.

1. Introduction

A constraint-satisfaction problem typically requires finding values for a set of variables, subject to an arbitrary set of constraints. For example, consider the problem of packing items into boxes, with the constraint that some pairs of items cannot be packed together (perhaps to avoid forming an explosive mixture). An acceptable solution to the packing problem is an assignment of items to boxes such that all constraints on pairs of items are satisfied. In a number of applications, one is interested in generating all acceptable solutions to a constraint-satisfaction problem. Many Artificial Intelligence and combinatorial search applications can be formulated as constraint-satisfaction problems.

A backtrack-search algorithm [*GOL*65] can be used to enumerate all solutions to a constraint-satisfaction problem. Previous work has primarily focussed on methods for improving the efficiency of backtrack search algorithms. Various authors have developed methods for improving search efficiency [*BIT*75] [*HAR*80] [*NUD*88]. The Iterative Deepening A* (IDA*) algorithm [*KOR*85] has been applied to find optimal cost solutions in AI applications modeled as state-space searches. Parallel execution methods for the IDA* algorithm are described in [*RAO*87] .

Most constraint-satisfaction problems are NP-complete and the algorithms used to solve them have worst-case exponential execution times. Using multiple processors to solve such problems cannot significantly improve the worst-case performance [*KAS*86] Nevertheless, since a large number of practical applications are naturally formulated as constraint-satisfaction problems, it is important to speed up their solution using one or both of the following approaches:

1. Develop efficient sequential algorithms for the average case.
2. Use multiple processors to speed up their execution by parallel processing.

In this paper, we are interested in parallelizations of the Forward-Checker algorithm [*HAR*80] which has been shown to be one of the most efficient sequential algorithms for constraint-satisfaction problems An efficient parallelization of the Forward-Checker algorithm, rather than the backtrack algorithm, gives us a true speedup over the sequential execution time.

We present two parallel algorithms that give near-linear speedups even beyond 100 processors. The first method is the Threshold Depth-First Priority (TDFP) algorithm which uses inter-node parallelism up to a specified granularity combined with a depth-first priority scheduling policy for tasks. The second method is the Breadth-First List Scheduling (BFLS) algorithm which uses inter-node parallelism to decompose the search tree into a fixed number of tasks (subproblems), and then applies the intra-node parallel algorithm from [*NAT*87] on each task simultaneously with a small number of processors per task. We present simulation results to support the following observations:

1. The TDFP algorithm is more robust with respect to imbalanced search trees; the algorithm can achieve near-linear speedup even when the imbalance becomes a sequential bottleneck for the BFLS algorithm.
2. The BFLS algorithm is more robust with respect to increasing multiprocessor overhead, while the performance of the TDFP algorithm degrades severely in this case.

An important property of our simulation is that we accurately measure the effect of scheduling overhead in the TDFP algorithm by treating a priority queue operation as a critical section. Thus, the parallel execution times measured include the effect of serialization in the task scheduler. Another important parameter studied is the effect of granularity. We show that, in the presence of overhead, there is an optimal intermediate threshold size which gives the best speedup.

The lookahead search in the Forward-Checker algorithm involves making a binding decision for a free variable, X, and discarding those values for the remaining free variables that are inconsistent with the value chosen for X. If all the remaining free variables still have a feasible value, the binding for X is performed and the search procedure moves forward in an attempt to find a binding for the next free variable. A solution is found when no free variables remain. However, if any of the remaining free variables has no feasible value, the binding for X is undone, and another value is chosen. When there are no more values, the search procedure backtracks to the previous variable. The reader is referred to [*HAR*80] for a detailed description of the sequential algorithm.

The rest of this paper is organized as follows. In Section 2, we describe two methods for parallelizing the sequential Forward-Checker algorithm. In Section 3, we present experimental results based on simulations of these algorithms. The experiments take into account the effects of scheduling, synchronization and serialization in executing the parallel algorithms. We present experimental results for two constraint-satisfaction problems, the N-Queens and Graph Coloring problems. In Section 4, we present our conclusions.

2. Processor Scheduling Algorithms

We assume a shared-memory multiprocessor model in this paper. Our simulation systen measures the parallel execution times, according to the task scheduling algorithm being used. The primary inter-processor interaction in the TDFP algorithm is due to insertions and deletions in the priority queue, which is accurately modelled as a critical section in our simulations. The priority queue contains all the necessary synchronization, since a parent task ensures that all the data needed by its child tasks is ready before the child tasks are created. Therefore, all other inter-processor interactions are secondary effects mainly due to interference among independent shared memory accesses. These effects depend on the shared memory architecture and are ignored in our simulations.

Ideally, a good parallel algorithm for a given problem should have the following desirable properties:

1. Sufficient parallelism for the number of processors we are interested in using.
2. Low overhead component in the parallel execution time. The overhead comes from controlling the parallel execution of a program (task creation, synchronization, communication, etc.).
3. Good load balancing of tasks on processors.
4. Efficient sequential execution time.
5. Reasonable space requirement. For our purpose, $O(NumProcs \times SeqSpace)$ space is reasonable, though $O(SeqSpace)$ space is ideal, where *NumProcs* is the number of processors and *SeqSpace* is the space used by an efficient sequential algorithm. $O(NumProcs \times SeqSpace)$ space can be considered reasonable for multiprocessors in which the total memory size increases linearly with the number of processors.

Properties 1, 2, and 3 are all necessary for good speedup. Property 4 is desirable for good performance measured in absolute terms. Property 5 is a necessary resource constraint to ensure that the parallel algorithm will execute with the available amount of memory. We next present two different algorithms for scheduling multiprocessor constraint-satisfaction searches and evaluate them against the criteria outlined above.

2.1 Threshold Depth-First Priority Algorithm

The Threshold Depth-First Priority algorithm (TDFP) described in this section satisfies all 5 properties as follows:

1. The TDFP algorithm exploits parallelism among nodes of the search tree. Since the number of parallel nodes could potentially be exponential, the amount of parallelism in the algorithm is more than adequate for any reasonable number of processors.
2. A low overhead component is obtained by using a threshold size to decide if a search tree node should be executed as a separate task or not. If the number of tests in the node is less than the threshold value then the node and all its descendants are executed sequentially as part of the parent task. Thus, the overhead of task creation, scheduling, synchronization, etc. only applies to computations larger than the threshold size, leading to a small amortized overhead. Naturally, this use of a threshold size reduces parallelism. But for the data sizes used in practice, there still remains plenty of parallelism after the threshold size has been made large enough to reduce the overhead component to an acceptable level.
3. Good load balancing is achieved by dynamically scheduling tasks at run-time. The scheduling policy is a form of non-preemptive list scheduling with no unforced idleness. This policy is guaranteed to yield a parallel execution time within a factor of 2 of the optimal schedule [*GRA*69].
4. The TDFP algorithm is a parallelization of the Forward-Checker search algorithm, which is considered to be one of the most time-efficient sequential algorithms for constraint-satisfaction problems. Both the parallel

TDFP and the sequential Forward-Checker algorithms perform exactly the same number of consistency tests. Hence our parallel algorithm has an efficient sequential execution time.

5. A depth-first priority rule is used to schedule tasks in the TDFP algorithm, so that nodes at a larger depth are executed first. This rule guarantees an $O(NumProcs \times SeqSpace)$ space usage, assuming that a parent task creates at most $O(NumProcs)$ child tasks. Space usage is a very important issue for tree-structured computations, where a simple FIFO scheduling policy could incur an exponential space expansion making it impossible for the program to continue execution.

The TDFP algorithm can be conveniently described by the recursive procedure, *Parallel__Search* , in Figure 1 which is structured like the version in [*NAT*87] The main difference is in the FORK statement. The WHEN clause in the FORK statement was introduced to avoid code duplication. If the WHEN condition is true, the computation enclosed in BEGIN...END is forked as a separate task with priority = *CurVar* If the WHEN condition is false, the same computation is just executed sequentially by the parent task.

This algorithm assumes that a scheduling mechanism exists in the multiprocessor system to assign each processor a task with the highest priority. It is necessary to maintain a run-time priority queue for scheduling tasks. Since the priority values must be in a small bounded range (up to the maximum depth of the search tree, say ≤ 100), a priority queue can be efficiently implemented as an array (indexed by priority values) of lists of tasks.

All data communication between parallel tasks in procedure *Parallel__Search* occurs between a parent and child. There are no global shared variables. The PRIVATE declaration gives each task its own copy of the variables. The TDFP algorithm thus has locality in communication. However, the priority-based task scheduler is assumed to be a centralized structure. An interesting area of future research would be to design efficient priority-based scheduling algorithms for distributed systems, which would allow the TDFP algorithm to work efficiently on message-passing multiprocessors as well.

Finally, we observe that this algorithm has no JOIN operation corresponding to the FORK's. So the parent task is just terminated after all its child tasks have been created. Thus, the parent does not need to be suspended till its child tasks have completed and the scheduling policy can be truly non-preemptive. Program execution is completed when there are no tasks in the priority queue and no tasks being executed on any processor.

```
PROCEDURE Parallel__Search(CurVar,F,FVT);
/* CurVar = current variable = computation depth */
/* F = partial solution, defined for F(1..CurVar-1) */
/* FVT is the Feasible Value Table. FVT(i) gives the
  list of feasible values for CurVar≤i≤NumVars. */
IF CurVar = NumVars THEN
  Output all solutions defined by F(1..CurVar-1)
  and FVT(CurVar)
ELSE
FOR V ← each value in FVT(CurVar) DO
/* If the number of tests in FVT(CurVar+1) ...
  FVT(NumVars) exceeds ThresholdSize, then fork
  BEGIN ... END as a separate task with priority=CurVar.
  Otherwise, execute the computation sequentially. */
  FORK WITH PRIORITY=CurVar
  WHEN size(FVT(CurVar+1..NumVars))
        ≥ ThresholdSize
  BEGIN
    PRIVATE New__F, New__FVT ;
    New__F ← F ; New__F(CurVar) ← V ;
    New__FVT ← Check__Forward(CurVar, V, FVT) ;
    IF New__FVT is not empty THEN
      Parallel__Search(CurVar+1,New__F,New__FVT)
  END
END FOR
END PROCEDURE

PROCEDURE Check__Forward(CurVar,CurVal,FVT);
/* Return a new FVT with those (Variable,Value)
  pairs that are consistent with (CurVar,CurVal).
  Return an empty table if some variable has no
  feasible value. */
/* Initialize NewFVT to an array of empty lists. */
NewFVT ← empty table ;
FOR FreeVar ← CurVar+1 TO NumVars DO
  FOR V ← each value in FVT(FreeVar) DO
    /* This is the consistency test. */
    IF (FreeVar,V) is consistent
      with (CurVar,CurVal) THEN
        Insert V in NewFVT(FreeVar)
  END FOR ;
  IF NewFVT(FreeVar) is empty THEN
  BEGIN
    NewFVT ← empty table;
    RETURN NewFVT
  END
END FOR ;
RETURN NewFVT
END PROCEDURE
```

Figure 1. Outline of the Threshold Depth-First Priority Algorithm

2.2 Breadth-First List Scheduling Algorithm

The Breadth-First List Scheduling Algorithm (BFLS) described in this section differs from the TDFP algorithm in how the multiple processors are allocated to search the tree. The algorithm exploits intra-node parallelism available at all levels of the search tree. The average amount of intra-node parallelism is large at nodes close to the root and falls off at deeper levels of the tree. If the amount of work to be done in a node is large enough to keep all the available processors busy, then all the processors are deployed to work in parallel within the node. However, when the available work within a node is not large enough to keep all the available processors performing useful work, then the algorithm explores many nodes in parallel (inter-node parallelism) by assigning a separate group of processors to each node.

The BFLS Algorithm was motivated by our earlier study [*NAT*87] where we observed a fall-off in intra-node parallelism with increasing node depth. The rationale for the present approach is to divide the search problem into smaller subproblems (or tasks) and to allocate the available processors to work in parallel on the tasks so that linear speedup behavior is achieved when a medium (100) to large (500) number of processors are used.

We next comment on the BFLS Algorithm with respect to the five points listed at the beginning of Section 2.

1. The parallelism used by the algorithm is limited by intra-node parallelism and by a user definable parameter *NumTasks*, the number of search subproblems created in Phase 1 of the algorithm.
2. The overhead is proportional to *NumTasks*. We ensure that *NumTasks* < *NumProcs*, so that the overhead will be *O*(*NumProcs*). If *NumTasks* is too small (close to 1), the algorithm will suffer from lack of parallelism. If the *NumTasks* is too large, the overhead will become significant. We have found that a reasonable choice for *NumTasks* is between 5% and 10% of *NumProcs*.
3. Good load balancing is achieved by the list scheduling algorithm used in Phase 2.
4. The BFLS algorithm performs all the consistency tests performed by the sequential Forward-Checker algorithm, and may also do some extra tests. The experimental results reported in Section 3.2 take into account any extra work that may be performed by the parallel algorithm.
5. The space requirement of the algorithm is *O*(*NumTasks* × *SeqSpace*), where *SeqSpace* is the space required by the sequential Forward-Checker algorithm. By specifying *NumTasks* to be *O*(*NumProcs*), we ensure the space required is reasonable.

The BFLS Algorithm is outlined in Figure 2. It consists of two phases. In Phase 1, a list of tasks corresponding to the subproblems is created. This is achieved by calling *Par__Ch__Forward*, a procedure that performs lookahead

```
/* Main Program */
/* N = Number of variables */
/* NumProcs = total number of processors */
/* FVT = Table of feasible values */
/* Phase 1: Creates a shared list of tasks */
  Create__Tasks (NumProcs,NumTasks,L);
ProcsPerTask = max (1,NumProcs / NumTasks)
/* Phase 2: Schedule tasks using List Scheduler */
while (L non-empty & a processor group is idle) do
 Schedule the next search Task L(i) for execution;
 Initialize F and FVT tables corresponding to L(i);
 INITIATE (BF__Par__LAS, L(i),ProcsPerTask);
 end;

 /* Phase 1 */
 Procedure Create__Tasks(NumProcs,NumTasks,L);
 /* Create list L : L(1), L(2), ... , L(NumTasks) */
  Par__Ch__Forward with NumProcs; Visit in
  breadth-first sequence and enqueue tasks in L until
  number of entries in L reaches NumTasks.

 Procedure BF__Par__LAS (Curr,F,FVT,PGS);
 /* Phase 2: Use Parallel Lookahead search within */
 /* each search task (see NAT87). Use PGS procs. */
 /* in parallel within a node. Array F stores    */
 /* values assigned to the bound variables.      */
 FOR F(Curr) = each element of FVT(Curr)
 DO BEGIN
  IF Curr < N
  THEN BEGIN
   Par__Ch__Forward(Curr,F(Curr),FVT,New__FVT,PGS);
   IF New__FVT not empty
   THEN BF__Par__LAS (Curr+1,F,New__FVT,PGS);
     END
  ELSE  Output the solution F;
  END BF__Par__LAS;

 Par__Ch__Forward(Curr,L,FVT,New__FVT,PGSize);
 /* New__FVT is New Feasible__Value Table. L is bound
 to Curr, FVT is revised and New__FVT is returned.
 PGSize processors are simultaneously used in
 the parallel execution of the nested do loops that
 perform the consistency tests.                */
   New__FVT := emptytable;
   FOR free__var := Curr+1 to N DOALL
    BEGIN
     FOR tvalue := each element of FVT(free__var)
     DOALL
      Perform lookahead tests in parallel (NAT87)
     ENDDO;
    ENDDO;
   return(New__FVT);
 END Par__Ch__Forward;
```

Figure 2. Outline of the Breadth-First List Scheduling Algorithm

operations within a node in parallel. In this invocation, all *NumProcs* processors are used to work in parallel within a node. After the list is created, the algorithm calculates *ProcsPerTask*, the number of processors to assign each task.

Phase 2 of the main program consists of the parallel execution of tasks in list L. A scheduler accesses list L in an exclusive mode, picks up the next task waiting to be executed and initiates a group of *ProcsPerTask* processors to execute it. Task initiation consists of invoking *BF__Par__LAS* , and passing it appropriate data structures corresponding to the task being scheduled. *BF__Par__LAS* in turn invokes *Par__Ch__Forward* to explore the nodes using *ProcsPerTask* processors within each node. Note that once a task is initiated, the scheduler is free to assign another task to another idle group of processors, provided such a group exists. If all processor groups are busy, the scheduler waits until a group becomes idle and assigns that group another task. The completion of Phase 2 occurs when all the tasks in list L have finished execution.

3. Experimental Results

In the following subsections, we present simulation results for the TDFP and BFLS algorithms. Only consistency tests are considered as useful work, since they dominate the computation in a search algorithm. The simulated execution times are normalized with respect to the execution time of a consistency test, so that each test is has execution time = 1. Results are presented for the following constraint-satisfaction problems:

1. *N*-Queens problem — Find all placements of *N* queens on an $N \times N$ chessboard such that no queen can attack another. Our experiments used $N = 8$.
2. *m*-Colorings of graphs — Given an undirected graph and *m* colors, find all possible ways in which the graph can be colored such that no two adjacent vertices in the graph are assigned the same color. Our experiments used $m = 5$ on a graph with 10 vertices (Figure 3). Vertex 1 was chosen as the distinguished vertex (fixed color). Note that the packing problem described in Section 1 can be formulated as a graph coloring problem.

3.1 TDFP Algorithm

A simulation of the Threshold Depth-First Priority algorithm (Section 2.1) was implemented with the following parameters:

1. *NumProcs*, the number of processors.
2. *ThresholdSize*, the threshold size specified as a minimum number of tests to be processed in a parallel task.
3. *QTime*, the overhead incurred by a processor to insert or remove a task from the priority queue. An important feature of our simulation is that the overhead time is assumed to be a critical section, just as in a real implementation. So, the waiting time to access the priority queue is often much more significant than *QTime*. In our simulation, the value of *QTime* is normalized with respect to the execution time of a consistency test (i.e. *QTime*=1 means that an enqueue or dequeue operation takes the same time as a test).
4. The search problem to be solved. Any constraint-satisfaction problem can be solved by this algorithm by appropriately initializing FVT and defining the consistency test predicate.

Figures 4a, 4b, 4c and 5a, 5b, 5c display results of experiments performed on the 8 Queens and Graph Coloring problems. The parallel execution time can be considered to have 4 components, corresponding to the 4 different states that a processor can be in:

$$T_{Par} = T_{Useful} + T_{WorkWait} + T_{QWait} + T_Q$$

where

1. T_{Useful} is the average time spent by a processor doing useful work (i.e. consistency tests).
2. $T_{WorkWait}$ is the average time spent by a processor waiting for work when the priority queue is empty. This usually occurs at the start of the program when a small number of tasks is available in the queue.
3. T_{QWait} is the average time spent by a processor waiting to enter the critical section of the priority queue. A processor needs to wait before it can pick a task for execution or insert any new child tasks in the queue.
4. T_Q is the average time spent by a processor on priority queue operations. If a processor needs to insert t tasks and then dequeue one task, it spends $(t + 1)QTime$ time as overhead after entering the critical section.

Figures 4a and 5a show the effect of threshold size on speedup for the 8 Queens problem on 100 processors and the Graph Coloring problem on 500 processors. For each value of *ThresholdSize* on the X-axis, the corresponding speedup was plotted on the Y-axis. The speedup is defined to be the ratio T_{Seq}/T_{Par}, where

- T_{Seq} is the total number of tests performed by the sequential Forward-Checker algorithm. Its value for the 8 Queens and Graph Coloring problems was 13024 and 202857 respectively.
- T_{Par} is the simulated parallel execution time.

There are 4 curves in Figures 4a and 5a corresponding to 4 different values of queue overhead, $QTime = 0, 0.1, 1, 10$. We define *OptThresholdSize*(*NumProcs*,*QTime*) to be that value of *ThresholdSize* which gives the largest speedup on

NumProcs processors with a queue overhead of *QTime*.[1] From Figure 4a for the 8 Queens problem, we can see that *OptThresholdSize* (100, QTime) = 1, 12, 23, 25 for *QTime* = 0, 0.1, 1, 10 respectively. From Figure 5a for the Graph Coloring problem, we have *OptThresholdSize* (100, QTime) = 1, 13, 17, 22 for *QTime* = 0, 0.1, 1, 10 respectively.

QTime = 0 represents the ideal zero overhead situation where the smallest threshold size (= 1) is optimal. In this case, the speedup only decreases (or stays the same) as the threshold size increases. However, the speedup curves for *QTime* = 0.1, 1, 10 all exhibit a maximum at an intermediate threshold size. This is the trade-off between overhead and parallelism. If the threshold size is too small, the overhead component increases causing the speedup to decrease. If the threshold size is too large, there is less parallelism again causing the speedup to decrease. This existence of an optimal granularity has been predicted for parallel programs in general [*SAR*86], [*SAR*87] and is observed here in a real application.

Figures 4b and 5b show how the parallel execution time from Figures 4a and 5a (for *QTime* =0.1) is split into its components. The 3 curves plotted are:

1. Useful Work Fraction = T_{Useful}/T_{Par}.
2. Work Wait Fraction = $T_{WorkWait}/T_{Par}$.
3. Queue Wait Fraction = $T_{QueueWait}/T_{Par}$.

The ratio T_Q/T_{Par} was not plotted because it was negligible ($<$ 0.01) compared to the other 3 components. The waiting time, T_{QWait}, is much more significant than the actual time spent in the critical section, T_Q. The threshold size that gives the maximum speedup can now be identified as that value which maximizes the Useful Work Fraction. The Queue Wait Fraction decreases (or remains the same) as *ThresholdSize* increases, which confirms that increasing the threshold size reduces the overhead component. Similarly the Work Wait Fraction increases (or remains the same) as *ThresholdSize* increases because of loss of parallelism.

The only exception to the monotonic behavior of the Queue Wait and Work Wait fractions occurs in Figure 5b when *ThresholdSize* goes from 4 to 5. We see an unexpected drop in the Work Wait Fraction with a corresponding rise in the Queue Wait Fraction. However, the actual value of T_{QWait} (rather than the Queue Wait Fraction) decreases montonically through 7159.4, 2319.8, 1761.4 for *ThresholdSize* = 3, 4, 5. The problem is in $T_{WorkWait}$ which goes through the values 619.9, 1253.4, 455.8. This erratic behavior in $T_{WorkWait}$ occurs because the priority queue is a critical section. At any given time, the queue contains a subset of the tasks that are ready to execute, because the processors must wait to enter the critical section before they can enqueue their newly created tasks. This subset can be significantly smaller than the set of ready tasks when *ThresholdSize* is small because of the longer waiting time to access the queue.

Finally, Figures 4c and 5c show the speedup as a function of the number of processors for *QTime* = 0, 0.1, 1, 10. The threshold size used for a given *QTime* value is that size which gave the best speedup on 100 processors for 8 Queens, or on 500 processors for Graph Coloring. We see that all 4 speedup curves begin with a linear increase and then flatten out, depending on the value of *QTime*. The speedup is very sensitive to *QTime*, which indicates that a real implementation should try and reduce the queue overhead as much as possible. There are a few points in Figure 4c where increasing the number of processors actually causes a small decrease in speedup. This occurs because our scheduling policy is not optimal. However, since we use a form of list scheduling, the theorem in [*GRA*69] guarantees that this anomalous decrease in speed-up cannot be by more than a factor of 2. This anomaly was only observed for 8 Queens, because the Graph Coloring problem contains more work and used a larger number of tasks which had a smoothing effect on the speedup curves.

For the 8 Queens problem on 100 processors, the speedups obtained for *QTime* = 0, 0.1, 1, 10 were 58.9, 49.9, 27.0 and 10.4 (see Figure 4c). For the Graph Coloring problem on 500 processors, the speedups obtained for *QTime* = 0, 0.1, 1, 10 were 364.2, 249.3, 78.3 and 35.5 (see Figure 5c). This establishes that the Threshold Depth-First Priority algorithm can attain reasonable speedups in the presence of low overhead. In fact, the simulated speedup values presented here are conservatively low. We can expect better speedups in a real implementation because:

- The problem size used in a real multiprocessor will be much larger than 8 Queens or Graph Coloring (which take around 1 CPU second on a mainframe). Therefore, a larger threshold size can be used to get the same amount of parallelism, but with a lower overhead component.
- An efficient implementation of the priority queue can reduce overhead and increase speedup in two ways:
  1. By reducing *QTime* , which would give a larger speedup.
  2. By further reducing T_{QWait} by allowing many processors to simultaneously update the queue.

  For example, if the queue is implemented as an array of lists of tasks indexed by priority values, then each queue operation takes constant time and the insertion and deletion of tasks with different priorities can proceed simultaneously.

[1] If there is more than one optimal value for *ThresholdSize*, let *OptThresholdSize*(*NumProcs*,*QTime*) be the smallest one.

3.2 BFLS Algorithm

A simulation of the Breadth-First List Scheduling Algorithm (Section 2.2) was implemented with the following parameters:

1. *NumProcs*, the number of processors.
2. *NumTasks*, the number of tasks created in the list from which tasks are assigned by the scheduler for processing by small groups of processors. For the purpose of this study the number of tasks created was controlled by an input parameter *BF__Upto__Level* which specified that the search tree be explored breadth-first at levels 1 through *BF__Upto__Level*. For depths greater than *BF__Upto__Level* , the search was executed according to the list scheduling algorithm.
3. *ProcsPerTask*, the number of processors per task was varied from 1 through 20.
4. The search problem to be solved.

In our simulation of the BFLS algorithm, the overhead associated with accessing the shared list of tasks was assumed to be zero (a reasonable assumption if task granularity is sufficiently large). In the next section, we estimate the effect of overhead due to list accesses on speedup.

Figure 6 displays results of experiments performed on the 8-Queens problem. We plot three curves showing Speedup vs Number of processors that reflect the effects of different combinations of breadth-first and list scheduling of search effort. Specifically, the curves correspond to the following cases:

- Using a value of *BF__Upto__Level* = 1, a small number of subproblems (8 tasks) were created.
- Using a value of *BF__Upto__Level* = 2, a medium number of tasks (42) were created.
- Using a value of *BF__Upto__Level* = 3, a large number of tasks (140) were created.

We make the following observations from Figure 6. When the number of tasks is small (= 8, a limited use of breadth-first search), the speedup increases upto about 100 processors and then flattens out at a limiting speedup value of 52 for larger number of processors. When a medium number of tasks (= 42, an intermediate amount of breadth-first search) is created, the speedup increases upto about 200 processors and then flattens out gradually at 125 (significantly higher than 52). When a large number of tasks (= 140, a large amount of breadth-first search) is created, the speedup tends to flatten out at about 62. This set of experiments suggests that the limiting speedup behavior of Breadth-First List Scheduling improves with increasing the number of tasks (i.e., the breadth-first component) up to some intermediate point and then starts deteriorating as the number of tasks is increased even further. The reason why the limiting speedup peaks at some intermediate number of tasks rather than increase monotonically with increasing number of tasks is as follows. The total time in executing the search consists of two components:

1. A Breadth-First component, where nodes of the search tree are examined sequentially and tasks are created for parallel execution.
2. A List Scheduling component, different tasks are executed in parallel.

The time spent in the Breadth-First component increases with the number of tasks created. This is shown in Figure 7 as $T_{Serial}(BF)$. The time spent in the List-Scheduling component decreases with increasing number of tasks created because of greater potential for inter-node parallelism. This is shown in Figure 7 as $T_{Parallel}(LS)$. There exists an intermediate value for the number of tasks such that the sum of the two components reaches a minimum (the right most bar in each set of three bars shown in Figure 7). Since the total amount of work done by the parallel algorithm is fixed, the work performed during the creation of a shared list of tasks must be balanced against potential benefit due to parallel execution of tasks.

Figure 8 shows results corresponding to Graph Coloring problem. Note that the graph shown in Figure 3 was used as an instance of the problem. Figure 8 displays results (Speedup vs Number of processors) of experiments performed with different task decompositions. Specifically, the following cases were considered:

- Using a value of *BF__Upto__Level* = 3, a small number of tasks (64) were created.
- Using a value of *BF__Upto__Level* = 4, a medium number of tasks (204) were created.
- Using a value of *BF__Upto__Level* = 5, a large number of tasks (816) were created.

When the number of tasks created is 64, speedup tends to flatten out at 229. When the number of tasks created is 204, the speedup tends to flatten out at about 413 (significantly more than 229). When the number of tasks created is 816, the speedup tends to flatten out at about 174.

The experiments performed with the N-Queens and Graph Coloring problems suggest that depending on the problem and its size, an optimal mix of sequential breadth-first traversal and parallel list scheduling should be used to maximize the performance benefit due to parallelism.

4. Conclusions

This paper has presented two new methods for parallelizing the Forward-Checker algorithm, namely the Threshold Depth-First Priority (TDFP) and the Breadth-First List Scheduling (BFLS) algorithms. Experimental results show that these algorithms can achieve near-linear speedups beyond 100 processors for problems where the speedup obtained by just using intra-node parallelism falls off at 10 processors.

The performance of the TDFP algorithm is very sensitive to the queue overhead, QTime, especially since a queue operation is assumed to be a critical section. It is very important to reduce the size of the critical section, and hence the value of QTime, as much as possible. For a priority queue implemented as an array (indexed by priority values) of lists, serialization is only necessary for operations with the same priority. Ideally, the basic FIFO queue operation for a given priority could just be a single Fetch-and-Add instruction (or some other equivalent instruction). While it is important to reduce the size of the critical section, the real solution to the serialization problem is to implement a distributed priority scheduler. This is an interesting area for future research.

An important parameter of the TDFP algorithm is the threshold size which defines the granularity of execution. The optimal granularity depends on the nature of the search problem, the number of processors (*NumProcs*) and the queue overhead (*QTime*). We observed that, in the presence of overhead, there is an optimal threshold size which gives the best speedup. This is the trade-off between overhead and parallelism. If the threshold size is too small, the overhead component increases causing the speedup to decrease. If the threshold size is too large, there is less parallelism again causing the speedup to decrease.

Similarly, the parameter which controls the granularity of the BFLS algorithm is *NumTasks*, the number of subproblems to be created. If the specified value is too small, the algorithm will suffer from lack of parallelizable work. If the specified value is too large, the sequential breadth-first traversal will become a bottleneck.

The TDFP and BFLS algorithms were found to perform well under different conditions. A speedup comparison is given in Table 1. The BFLS speedups were obtained assuming 42 tasks for 8 Queens and 204 tasks for Graph Coloring (i.e. the best values for *NumTasks*). For the BFLS algorithm, the speedup with overhead was simply computed as $NumTests/(T_{Par} + 2 \times NumTasks \times QTime)$, so that each task incurs an enqueue and a dequeue overhead just as in the TDFP algorithm. A * is placed against a speedup entry in the table if the speedup is at least 10% more than the speedup due to the other method.

For low overhead (QTime = 0, 0.1), the TDFP algorithm performed better than the BFLS algorithm for the Graph Coloring problem and they performed comparably on the N-Queens problem. This is because the search tree for Graph Coloring problem is more imbalanced than the N-Queens problem. For large overhead (QTime = 1, 10), the BFLS algorithm performed better than the TDFP algorithm on both problems. This is because the TDFP algorithm is more sensitive to increased overhead. Therefore, both algorithms are suitable for solving constraint-satisfaction problems. The best choice depends on the search problem and the overhead in the target multiprocessor.

References

[BIT75] J. R. Bitner and E. M. Reingold, "Backtrack programming techniques," *Comm. of the ACM*, vol. 18, pp. 651-656, Nov 1975.

[GOL65] S. Golomb and L. Baumert, "Backtrack programming," *Journal of the ACM*, vol. 12, 1965, pp. 516-524.

[GRA69] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics* 17(2), March 1969.

[HAR80] R. M. Haralick and G. L. Elliott, "Improving tree search efficiency for constraint-satisfaction problems," *Artificial Intelligence* pp.263-313, 1980.

[KAS86] S. Kasif, "On the parallel complexity of some constraint-satisfaction problems," *Proc. of the Fifth National Conference on Artificial Intelligence*, Philadelphia, PA, 1986, pp.349-353.

[KOR85] R. E. Korf, "Depth-first iterative-deepening: an optimal admissible tree search," *Artificial Intelligence*, 1985.

[NAT87] K. S. Natarajan, "An empirical study of parallel search for constraint-satisfaction problems," IBM Research Report 13320, Nov. 1987.

[NUD88] B. A. Nudel, "Tree search and arc consistency in constraint satisfaction algorithms," in *Search in Artificial Intelligence*, edited by L. Kanal and V. Kumar, Springer-Verlag, 1988 (to appear).

[RAO87] V. N. Rao, V. Kumar and K. Ramesh, "A parallel implementation of the IDA* algorithm," *Proc. of the AAAI-87,* July 1987, Seattle, WA.

[SAR86] V. Sarkar and J. L. Hennessy, "Partitioning Parallel Programs for Macro-Dataflow," *Proc. of the ACM Conference on Lisp and Functional Programming,* August 1986, pp. 202-211.

[SAR87] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors," Ph.D. thesis, Stanford University, April 1987, Technical Report No. CSL-TR-87-328.

Problem	Procs	QTime	Method	Speedup
8Queens	84	0.0	TDFP	53.8
8Queens	84	0.0	BFLS	54.3
8Queens	84	0.1	TDFP	49.0
8Queens	84	0.1	BFLS	52.4
8Queens	84	1.0	TDFP	27.0
8Queens	84	1.0	BFLS	40.2 *
8Queens	84	10.0	TDFP	10.4
8Queens	84	10.0	BFLS	12.1 *
Coloring	410	0.0	TDFP	314.0 *
Coloring	408	0.0	BFLS	237.3
Coloring	410	0.1	TDFP	230.9
Coloring	408	0.1	BFLS	226.5
Coloring	410	0.1	TDFP	78.3
Coloring	408	1.0	BFLS	160.6 *
Coloring	410	10.0	TDFP	35.5 *
Coloring	408	10.0	BFLS	41.1 *

Table 1: Speedup comparison of the TDFP and BFLS algorithms

Figure 4(a): **8 Queens: Effect of ThresholdSize**

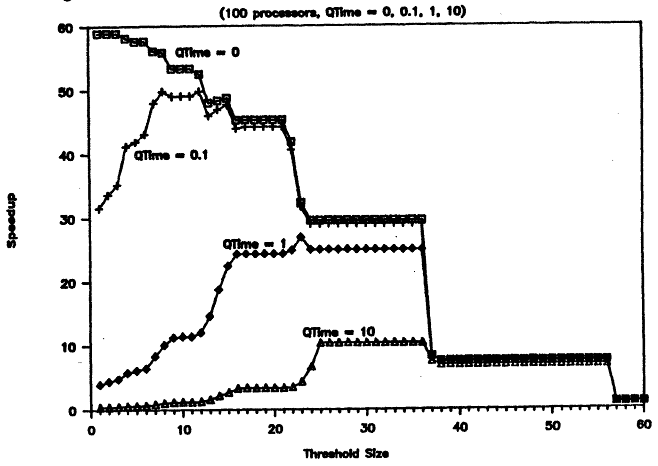


Figure 4(b): **8 Queens: Parallel time components**

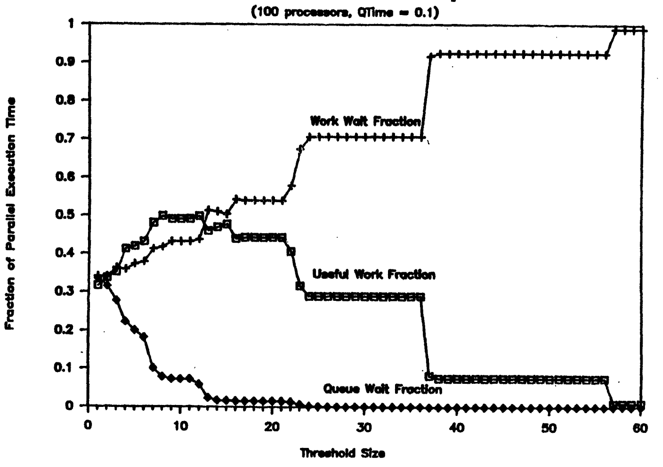


Figure 3: Graph used in Coloring Problem

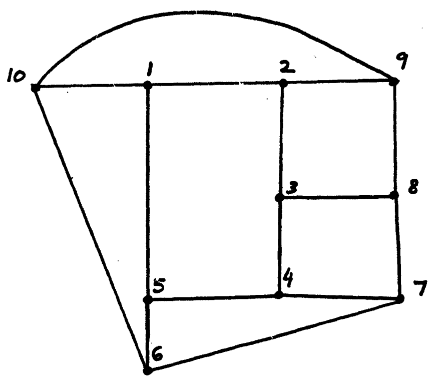


Figure 4(c): **8 Queens: Speedup vs Processors**

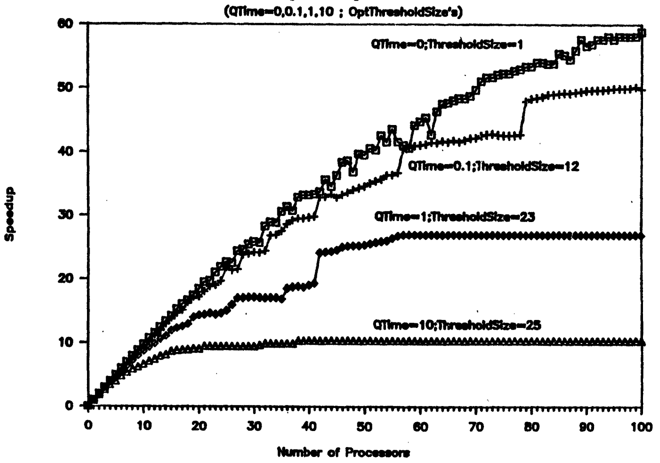


Figure 5(a): **Graph Coloring: Effect of ThresholdSize**

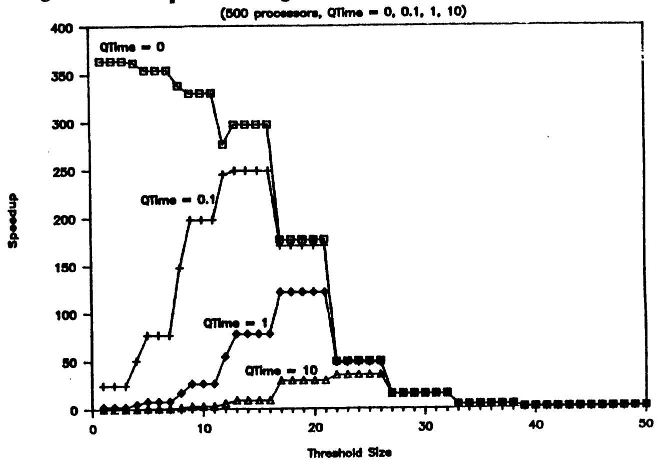


Figure 6: **8 Queens: Speedup vs Processors**

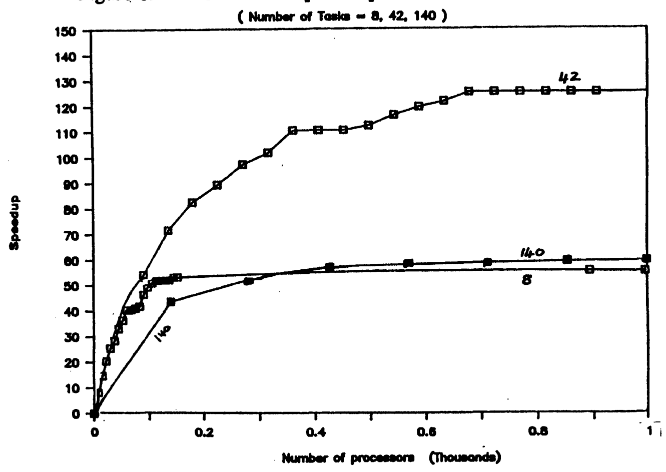


Figure 5(b): **Graph Coloring:Parallel time components**

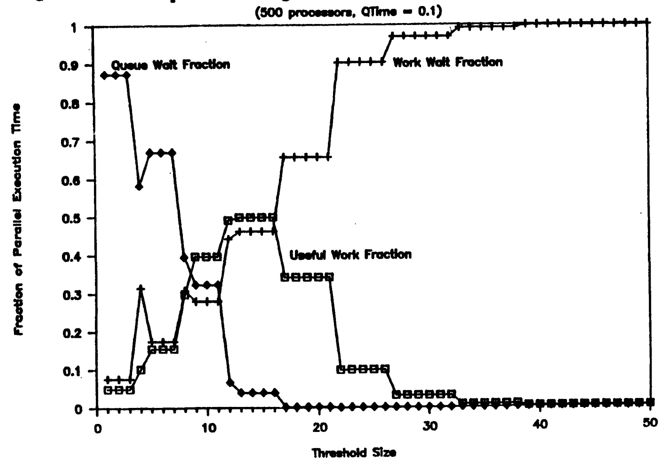


Figure 7: **Components of Total Time**

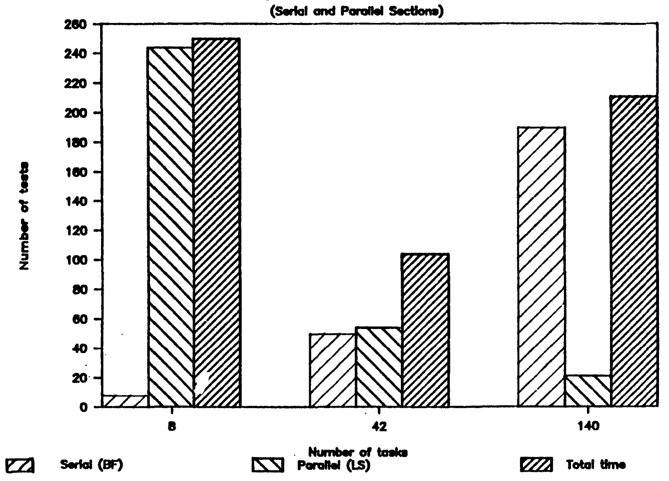


Figure 5(c): **Graph Coloring: Speedup vs Processors**

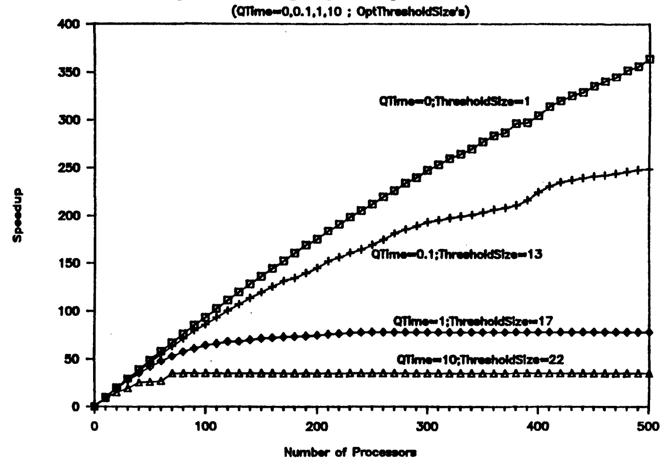
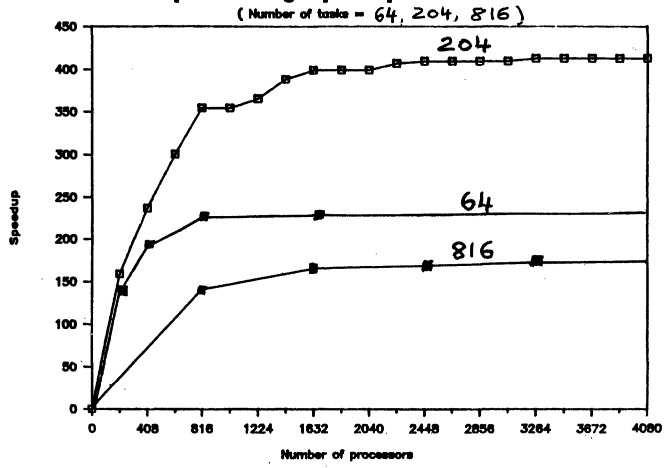


Figure 8 **Graph Coloring: Speedup vs Processors**



ON THE SEMANTICS OF PRIORITY SYSTEMS

Ryszard Janicki Peter E. Lauer

Department of Computer Science and Systems
McMaster University
1280 Main Street West, Hamilton, Ontario, Canada L8S 4K1

Abstract

The paper presents a formal definition of the semantics of concurrent systems with priority constraints. It is shown that neither partial orders, which form the basis of several formal semantical theories which explicitly model concurrency, nor total orders, which form the basis of all formal semantical theories which reduce concurrency to arbitrary interleaving, suffice for the treatment of priority. We use a dialect of COSY, a notation for specifying concurrent systems whose formal semantics is based on partial orders for the treatment of systems in the absence of priorities. We introduce a semantical theory of COSY systems including priority constraints which is based on sequences of multiples. Finally, we show how the concept of *starvation* can be formally defined in the approach presented.

1 Introduction

In non-sequential systems, **priority** denotes the order of preference in which events in conflict obtain service. It may be based on the nature of the service being requested or the importance of the events involved. Various techniques for specifying priorities in real-time systems together with their intuitive semantics, are discussed in [1]. Priorities are used to advantage to obtain COSY specification of a hyperfast banker's strategy in [2]. The programming language *occam*[1][3], a commercial version of Communicating Processes CSP [4], has a priority operator similar to the one used in COSY. However, CSP and its accompanying theory usually skirts the issue of priorities (see [5][4]). To develop a formal semantics for priorities is difficult and controversial, and a fully satisfactory formal solution has yet to be discovered in our opinion. Some reasons for problems arising in the formal treatment of priorities are described by Leslie Lamport in [6]. In the present paper we point out another source of problems and suggest a solution to it. The approach presented here is based in part on the results of [14].

2 Problem Explanation

Let us consider the following part of an *occam*-like program (see [3][5]):

```
...
PAR
  WHILE TRUE
    PRI ALT
      SEQ
        stop?ANY
        errorrecoveryandrestart
      read?x
  WHILE TRUE
    SEQ
      errormessage?ANY
      stop!ANY
...
```

The program above can be modelled by the following **extended** Theoretical Communicating Sequential Processes TCSP program (see [7][4]):[2]

$$\begin{aligned} Pr_1 = \ & \mu P.(stopandrestart \to P \ \Box_> \ readx \to P) \\ & \| \ \mu P.(errormessage \to stopandrestart \to P) \end{aligned}$$

where *readx* corresponds to *read?x*, *stopandrestart* corresponds to:

```
SEQ
  stop?ANY
  errorrecoveryandrestart
```

in the first loop, and to *stop!ANY* in the second loop (compare [5]), and *errormessage* corresponds to *errormessage?ANY*. The only extension we have allowed ourselves over standard TCSP is the use of the operator $\Box_>$, which corresponds to *PRI ALT* in *occam*, and it denotes the resolution of *non-deterministic* choice in favour of the left side. $P \Box_> Q$ means that any conflict between P and Q should always be resolved in favour of P, i.e. Q may be activated only if other synchronization constraints disallow activation of P. The *extended* Petri net in Fig.1 models both our part of the *occam*-like program, and the extended TCSP program (see [8], and [9], for the relationship between Petri nets and TCSP). Here, the only extension to standard Petri nets is that we attach a description of the priority $b > c$ to the place s_3. The notation $s_3(b > c)$ means that "b and c must be output transitions of s_3, and any conflict between b and c must be resolved in favour of b".

The standard semantics of Petri nets is defined in terms of labelled partial orders (see [10]), where partial orders represent possible histories of the system represented by a given net. Many arguments in favour of partial order semantics in

[1] *occam* is a trade mark of the INMOS Group of Companies.

[2] In this paper we use the notation of [4].

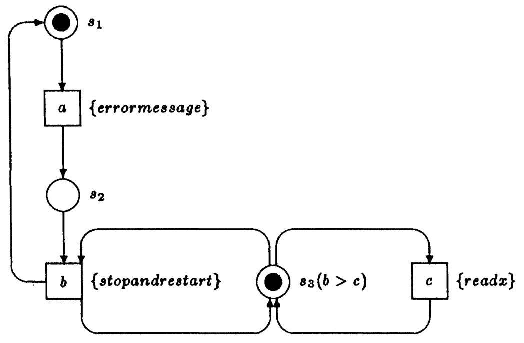


Figure 1: Extended Petri Net for program Pr_1

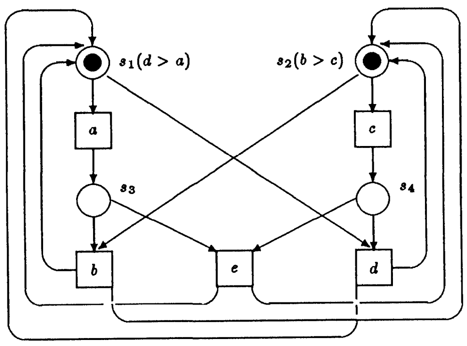


Figure 2: Extended Petri Net for Pr_2.

general, not just for Petri nets, can be found in [11]. Unfortunately, the above net shows that *partial order semantics does not work for our extended nets with priorities*. One may easily show that the sequence of transitions ca, and the simultaneous firing of transitions a and c, are both possible behaviours, while the sequence of transitions ac is *not*! The reason is that after firing a the conflict between b and c arises, and it has to be resolved in favour of b. The simultaneous firing of a and c is allowed, since initially there is no conflict between b and c, and b is not enabled until a has been fired. But this means that a history involving a and c *cannot be modelled by a partial (causal) order*. Suppose that there is a partial order which models such a history adequately. Because of the possibility of the simultaneous firing of a and c, there should be no order between a and c in this partial order. But this would imply that sequences ac and ca would be possible behaviours, whereas we have pointed out above that ac is not a possible behaviour. The standard semantics for TCSP is based on the notion of arbitrarily interleaving the independent events of concurrent subsystems to form one totally ordered history, thus reducing concurrency to arbitrary choice (see [4][7]), however partial order semantics may be used also (see [8][9][25]). The notion of interleaving seems to work for our example Pr_1. The set of all interleavings generated by Pr_1 can be described as:

$Pref((readx^*.(errormessage.stopandstart))^*)$

where $Pref(L)$ denotes the set of all prefixes of L, where L is a language (or a string set), and "." denotes string concatenation. For some arguments against interleaving semantics for concurrent systems see [8] [9][25]. However, the next example will show that we cannot use interleaving semantics to model priorities in general. Consider the following extended TCSP program:

$$\begin{aligned} Pr_2 = \; & \mu P.((d \to P)\Box_{>}(a \to (b \to P \;\Box\; e \to P))) \\ & || \; \mu P.((b \to P)\Box_{>}(c \to (d \to P \;\Box\; e \to P))) \end{aligned}$$

A corresponding extended Petri net is that of Fig.2. In both cases the set of interleavings generated is $Pref((ab \cup cd)^*)$. Thus according to interleaving semantics the event (transition) e *never occurs*. For e to occur, a and c must occur *simultaneously*. Furthermore, this implies that both Pr_1 and Pr_2 are *not live* according to interleaving semantics, while both are *live* according to intuition. It is also possible to give non-artificial examples of *occam*-like programs for which interleaving semantics does not work correctly, when compared to the intuitive understanding of the intended meaning of the program. We refrain from giving more elaborate examples here due to lack of space and the need to introduce more *occam* constructs.

3 Towards a Solution

In our approach we shall use sequences of multiples to mathematically model system behaviour, and represent the concurrent systems we will be modelling in a COSY dialect which is suited to the task.

3.1 Observations and Multiple Sequences

A partial order representing a history can be interpreted as a causality relation holding between event occurrences (or transition firings). No order between two event occurrences means that they are regarded as causally independent. Interleaving semantics is based on the assumption that an observer can observe only one event occurence at a time. However, we can reason about behaviour in another way. Assume that a possible observer of system behaviours can *only* detect either *sequential* or *simultaneous* occurrences of events at any one point (see [12]). Under such an assumption every observation can be represented as a sequence of sets of events. For instance, the observation:

> "first, an observation of simultaneous occurences of a, b and c; followed by an observation of an occurence of d alone, followed by an observation of simultaneous occurences of a and e"

can be represented by a sequence of multiples as follows (see [13][14]):

$\{a, b, c\}\{d\}\{a, e\}$

where $\{a, b, c\}$ denotes a simultaneous occurrence of a, b and c, $\{d\}$ denotes an occurrence of d alone, and $\{a, e\}$ denotes a simultaneous occurrence of a and e. Each such sequence will be called a **multiple sequence**. Multiple sequence semantics is a generalization of interleaving semantics in that more observational power is given to observers. Multiple sequences can also be interpreted as a kind of partial order in the sense that interleavings can be interpreted as total orders (see Fig.3). However, partial orders defined by multiple sequences

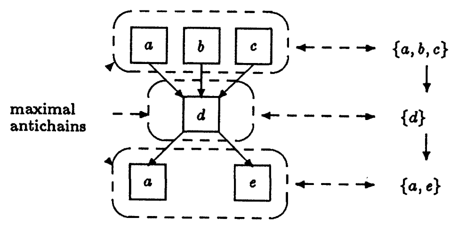


Figure 3: Partial (but not causal) order corresponding to $\{a,b,c\}\{d\}\{a,e\}$.

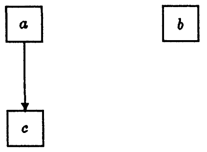


Figure 4: Partial (causal) order (invariant, partial order scheme.

are very special, and in general they cannot be interpreted as histories of causally related event occurrences.

For these orders, maximal antichains correspond to multiples (i.e. sets of simultaneous event occurrences), *maximal antichains are always disjoint*, and *each such order can always be represented as a total order of its maximal antichains* (see Fig.3). This is a simple consequence of the fact *that in this approach simultaneity is transitive when treated as a relation, whereas concurrency is clearly not transitive.* A partial (causal) order representing a system history can be interpreted as an **invariant** (or *partial order scheme*), i.e., the maximal order which is satisfied by every observation. This interpretation is also possible in the interleaving model. For systems whose histories could be modelled by partial (causal) orders, an invariant represents all possible behaviours and vice versa. For instance, a history represented by the partial (causal) order from Fig.4. can equivalently be represented by the following multiple sequences (observations):

$$\{a\}\{b\}\{c\}, \{a\}\{c\}\{b\}, \{b\}\{a\}\{c\}, \{a,b\}\{c\}, \{a\}\{b,c\}$$

For systems with priorities some observations (multiple sequences) are not allowed, although they can satisfy a given invariant, so usually in this case, an *invariant describes more observations than are really possible.* However, we can use multiple sequences (observations) to represent non-sequential behaviours directly, as it is done in the interleaving approach with regard to concurrency. The diagram in Fig.5 represents the behaviours of the system specified by the extended TCSP program Pr_1 and the extended Petri net from Fig.1, and the diagram in Fig.6 represents the behaviours of the system specified by the extended TCSP program Pr_2 and the extended Petri net from Fig.2, and their behaviours seem to be exactly as we intuitively expect. Unfortunately, the multiple sequence approach has some disadvantages. But it *works for systems with priorities.* The assumption that an observer can only observe **either** sequence **or** simultaneity disallows some observations. For instance, in the case of the partial order examined immediately above and depicted in Fig.4, it disallows

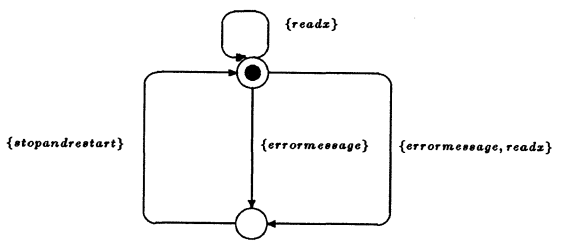


Figure 5: The observation behaviour of Pr_1.

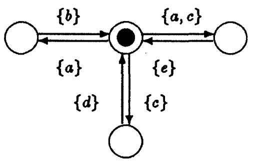


Figure 6: The observation behaviour of Pr_2.

the observation:

> "a simultaneous observation of an occurrence of b and of the sequence of occurrences ac,"

that is, something like $\{b, ac\}$, although the partial (causal) order describes such a possibility as well. Giving more power to observers will result in an extremely complex formal theory. In some cases, the precise interpretation of "simultaneous occurrences" of events is not obvious and might be ambiguous. All the problems dissappear when one uses partial (causal) orders, so in many respects the partial order approach (as in Petri net theory, standard COSY semantics, or in [11]) is better, *if applicable.*

3.2 Multiple Sequence Semantics of a COSY dialect with Priorities

COSY (short for **CO**ncurrent **SY**stem) is an abstract notation for specifying the synchronization properties of concurrent systems in abstraction from all other properties of the system, such as which mathematical functions a system may be computing, or what other interpretations basic events of the system may have. It can be seen as an extension of regular expressions to include parallelism (in the same sense as Petri nets can be regarded as an extension or "product" of finite automata). The COSY notation was introduced in [15], and subsequently was extended to a high level notation to express common structures of subsystems succinctly, and for obtaining parametrized specifications of systems, during a succession of research projects at the University of Newcastle upon Tyne, U.K., the GMD Research Establishment, Bonn, West Germany, and McMaster University, Canada (see [16][17][18][9]). Here we will only introduce the bare essentials to make our points. Similar use of uninterpreted symbols to denote events form the basis of CCS [12] and TCSP [7].

Let Ev be an arbitrary (but finite) set of events, P and P_i denote sequential COSY subsystems, and Pr denote a COSY system composed of such subsystems. The following BNF-style grammar defines an abstract syntax for our COSY dialect:

$$P \ ::= \ a \ / \ P_1 ; P_2 \ / \ P_1 , P_2 \ / \ (P)*$$
$$Pr \ ::= \ P_1 \ || \ \cdots \ || \ P_n$$

where $a \in Ev$. The grammar above contains ambiguities which can be resolved by using parentheses. In the absence of parenthesis we use the convention that "," binds stronger than ";", and the latter binds stronger than "||". Hence, $a, b; c$ is equivalent to $(a, b); c$. Instead of $(a)*$ we can also write $a*$. The semicolon denotes sequence (concatenation), the comma denotes mutually exclusive choice, and $*$ denotes the Kleene star, i.e., $(P)*$ means that P may be executed zero or more times. Syntactically, P is nothing but a kind of regular expression, although they are interpreted differently. The abstract grammar above is a modification of a grammar presented in [9]. The full COSY notation can be found in [16][18]. Its standard partial order semantics is defined on the basis of *vector firing sequences* [19]. For more details the reader is referred to [9][18].

Let $Pr = P_1 || \cdots || P_n$ be a COSY program. By Ev we denote the set of all events involved in Pr, and by Ev_i we denote the set of all events involved in P_i for $i = 1, \ldots, n$. That is, $Ev = Ev_1 \cup \ldots \cup Ev_n$. For any i, let $Seq(P_i)$ denote the set of all sequences (or language) over Ev_i defined by P_i interpreted as a regular expression. For instance:

$Seq(a, b; c) = \{ac, bc\}$ and $Seq((d; c)*; b) = \{dc\}^* \{b\}$.

For every set $L \subseteq Ev^*$, let $Pref(L) = \{x | (\exists y \in Ev^*) xy \in L\}$. The set of all possible observations of the behaviour of P_i, the **firing sequences** of P_i, denoted by FS_i, is defined by: $FS_i = Pref(Seq(P_i))$. From $FS_1, \ldots, FS_n$ we can derive the set of all histories (causal partial orders) generated by Pr (see [19][18][9] for details).

The relation $ind \subseteq Ev \times Ev$, called the **independency**, is defined as:

$$(\forall a, b \in Ev)(a, b) \in ind \Leftrightarrow a \neq b \wedge (\forall i)(a \notin Ev_i \vee b \notin Ev_i).$$

The relation *ind* defines independent events and only independent events can eventually (but not neccessarily) occur simultaneously.

Define a family of sets of events $Ind \subseteq 2^{Ev}$ as follows:

$$A \in Ind :\Leftrightarrow (\forall a, b \in A) a = b \vee (a, b) \in ind.$$

The elements of Ind are simply sets of mutually independent events.

Let $lc \subseteq Ev \times Ev$ be the following **local conflict** relation:

$$(a, b) \in lc :\Leftrightarrow a \neq b \wedge (\exists i)(\exists x \in FS_i) xa \in FS_i \wedge xb \in FS_i.$$

Note that:

$$(a, b) \in lc \Rightarrow (\exists i) a \in Ev_i \wedge b \in Ev_i \Leftrightarrow (a, b) \notin ind.$$

If $(a, b) \in lc$, i.e. if they are in a local conflict, then there is at least one sequential component of the whole system, which when considered on its own, after a sequence of event occurrences x, permits an occurrence of *either a or b* (but not both, since $(a, b) \notin ind$). Figure.7 illustrates the relations *ind*, *lc* and the family of sets of events Ind. Let $< \subseteq Ev \times Ev$ be a relation satisfying:

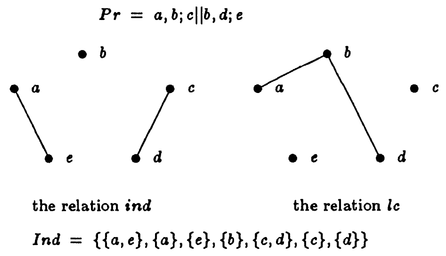


Figure 7: An illustration of *ind*, *lc* and *Ind*.

(1) $a < b \Rightarrow \neg(b < a)$

(2) $a < b \Rightarrow (a, b) \in lc$.

Condition (1) says that $<$ is asymetric, and Condition (2) indicates that $<$ can only be defined for events in a local conflict. The relation $<$ is called **priority** (we do *not* assume that $<$ is transitive). By a **priority COSY program** we mean a pair:

$$Pr = < \ P_1 || \ldots || P_n$$

where: $<= \{a_1 < b_1, \ldots, a_k < b_k\}$ is the definition of a priority relation $<$, and $P_1 || \ldots || P_n$ is a COSY program. For example:

$Pr_1 =$
$\{readx < stopandrestart\}$
$(errormessage; stopandrestart) * \ || \ (readx, stopandrestart)*$

corresponds to the TCSP program Pr_1 and the Petri net in Fig.1, and:

$$Pr_2 \ = \ \{a < d, c < b\}$$
$$((a; b, e), d) * \ || \ ((c; d, e), b)*$$

corresponds to the TCSP program Pr_2 and the Petri net in Fig.2.

Let ε denote both the empty sequence and the empty multiple sequence, let Ind^* denote the family of all multiple sequences over the alphabet Ind including ε, and for $i = 1, \ldots, n$, let $h_i : Ind^* \longrightarrow Ev^*$ be the following homomorphism:

$(i) \quad h_i(\varepsilon) = \varepsilon$

$$(ii) \quad (\forall A \in Ind) h_i(A) = \begin{cases} \varepsilon & \text{iff } A \cap Ev_i = \emptyset \\ a & \text{iff } A \cap Ev_i = \{a\} \end{cases}$$

The correctness of the definition of h_i follows from the fact that:

$$(\forall i = 1, \ldots, n)(\forall A \in Ind) cardinality(A \cap Ev_i) \leq 1.$$

Since h_i is a homomorphism:

$h_i(A_1 \ldots A_k) = h_i(A_1) \ldots h_i(A_k)$ for $A_1, \ldots, A_k \in Ind$.

A set $A \in Ind$ is said to be **potentially enabled** (i.e. without taking into account the priority constraints) at $t \in Ind^*$ if and only if:

$$(\forall i = 1, \ldots, n) h_i(tA) \in FS_i.$$

Let $mpenabled(t)$ denote the family of all sets (multiples) potentially enabled at t. A set $A \in Ind$ is said to be **enabled** at $t \in Ind^*$ if and only if:

(1) A is potentially enabled at t (i.e $A \in mpenabled(t)$),

(2) $(\forall a \in A)(\forall B \in mpenabled(t))(\forall b \in B)\neg(a < b)$.

The second condition says that there is no potentially enabled event which has a greater priority than elements of A. The family of all sets (multiples) enabled at t will be denoted by $menabled(t)$.

For example in the case of the COSY program Pr_1:

$mpenabled(\varepsilon) = menabled(\varepsilon)$
$= \{\{readx\}, \{errormessage\}, \{errormessage, readx\}\}$,

$mpenabled(\{readx\}) = menabled(\{readx\})$
$= \{\{readx\}, \{errormessage\}, \{errormessage, readx\}\}$,

$mpenabled(\{errormessage\}) = \{\{readx\}, \{stopandrestart\}\}$,

$menabled(\{errormessage\}) = \{\{stopandrestart\}\}$, so
$mpenabled(\{errormessage\}) \neq menabled(\{errormessage\})$,

$mpenabled(\{errormessage, readx\})$
$= menabled(\{errormessage, readx\})$
$= \{\{stopandrestart\}\}$, and so on .

Let $R_< \subseteq Ind^* \times Ind^*$ be the following relation:

$(\forall t, r \in Ind^*)\, tR_<r \;:\Leftrightarrow (\exists A \in menabled(t))\; r = tA.$

The relation $R_<$ is called the **reachability in one step.** The set of **possible observations**, the **multiple firing sequences** of Pr, denoted by MFS (or $MFS(Pr)$ if necessary), is defined by:

$MFS \;=\; \{t \mid \varepsilon R^*_< t\}.$

For the COSY programs Pr_1 and Pr_2 we have:

$MFS(Pr_1) = Pref((\{readx\}^*((\{errormessage, readx\}$
$\cup\{errormessage\})\{stopandrestart\}))^*)$,

$MFS(Pr_2) = Pref((\{a\}\{b\} \cup \{c\}\{d\} \cup \{a, c\}\{e\})^*).$

where we have written a regular expression over the alphabet Ind rather than the explicit description of the multiple language. Note that these are exactly the behaviours depicted in the last two diagrams. Let $\approx$ be the following relation on MFS:

$(\forall t, r \in MFS)\; t \approx r \;:\Leftrightarrow\; (\forall i = 1, \ldots, n)h_i(t) = h_i(r).$

One may easily prove that $\approx$ is an equivalence relation. Let $[t]_\approx$ be the eqivalence class of $\approx$ containing t. The interpretation of $\approx$ is the following. Suppose we have n local observers, and the i-th observer can only observe events belonging to Ev_i. If $t \approx r$ then t and r are different observations of the same history, and $[t]_\approx$ is the set of all observations of a history. Note that in general $[t]_\approx$ **cannot** be constructed from the $h_i(t)$'s alone, i.e., we cannot replace MFS by Ind^* in the definition of $\approx$.

The multiple firing sequence semantics may also be applied to ordinary COSY programs without priorities, simply by assuming that the priority relation is empty. In this case MFS can be described without the use of $R_<$, using only $FS_1, \ldots, FS_n$.

Theorem 3.1 [14]
If $<$ is the empty relation then
$MFS = \{t \in Ind^* | (\forall i = 1, \ldots, n)h_i(t) \in FS_i\}.$ ■

One immediate consequence of Theorem 3.1 is that if $<$ is the empty relation then we can replace MFS by Ind^* in the definition of $\approx$, since in such a case:

$[t]_\approx = \{r \in Ind^* | (\forall i = 1, \ldots, n)h_i(t) = h_i(r)\}.$

The COSY program Pr_2 shows that, if $<$ is not the empty relation then the behaviour of a system *might not be simulatable by interleavings* (e is never allowed to occur under interleaving semantics).

We say that $Pr =< P_1 || \ldots || P_n$ is **serializable** if and only if:

$(\forall t \in MFS \setminus \{\varepsilon\})(\exists a_1, \ldots, a_k \in Ev)\; \{a_1\} \ldots \{a_k\} \;\approx\; t.$

where "\" denotes set subtraction. The COSY program Pr_1 is serializable, but Pr_2 is not.

Theorem 3.2 [14]
If $(\forall t \in MFS)(\forall A \in menabled(t))(\forall B \subset A)$
$A \setminus B \in menabled(tB)$ *then Pr is serializable.* ■

The multiple firing sequence model of behaviour allows us to speak formally of dynamic properties of a system specified by a priority COSY program $Pr =< P_1 || \ldots || P_n$. We say that $Pr =< P_1 || \ldots || P_n$ is **deadlock-free** if and only if:

$(\forall t \in MFS) menabled(t) \neq \emptyset.$

We say that $Pr =< P_1 || \ldots || P_n$ is **adequate** (contains no partial system deadlock, see [18][19][9]) if and only if:

$(\forall t \in MFS)(\forall a \in Ev)(\exists r \in R^*_<(t))\; \{a\} \in menabled(r)$

where $R^*_<(t) = \{r \in Ind^* | tR^*_< r\}$. It can be proved that for $<$ equal to the empty relation the above notions correspond directly to the similar notions in the standard COSY approach used in [18], see [14].

4 Starvation and Infinite Multiple Sequences

Starvation occurs when some part of a system is constantly prevented from progressing. The reasons for starvation might be unfair conflict resolution, or conspiracy on the part of some subsystems against some other subsystem(s). Although the intuition at the basis of the notion of starvation seems rather clear, it turns out to be extremely difficult to formalize (compare [20]). To define starvation for COSY programs we need to define the concept of infinite behaviour.

Let $Pr =< P_1 || \ldots || P_n$ be a priority COSY program. The set of **infinite possible observations**, or the **infinite multiple firing sequences** of Pr, denoted by $IMFS$ (or $IMFS(Pr)$ if necessary), is defined by:

$IMFS =$
$\{A_1 \ldots A_i \ldots | (\forall i = 1, \ldots, \infty)A_i \in Ind \wedge A_1 \ldots A_i \in MFS\}.$

We say that $Pr =< P_1 || \ldots || P_n$ is a **simple request system for** $a \in Ev$, and $b \in Ev$ is **the request of** a **in** Pr, if and only if there is i_b, $i_b = 1, \ldots, n$, such that:

(1) $(\forall j)(i_b \neq j \Rightarrow b \notin Ev_j) \wedge (\forall c \in Ev_{i_b})(b,c) \notin lc)$,

(2) $(\forall x \in h_{i_b}(MFS))\ pr_{\{a,b\}}(x) \in (ba)^*(b \cup \varepsilon)$,

where in (2) the regular expression $(ba)^*(b \cup \varepsilon)$ represents a language as usual, and $pr_{\{a,b\}}(x)$ denotes x after erasing all events except a and b (projection of x onto $\{a,b\}$). If Pr is a simple request system for a, and b is the request for a, we shall write $req(a)$ instead of b.

The Condition (1) of the definition of simple request system says that $b = req(a)$, the request for a, occurs in one sequential component only, and it is not involved in any conflict. The Condition (2) says that the sequential component containing $b = req(a)$ has a special form, namely, if x is a finite firing sequence of this component, generated by a finite behaviour of Pr, and if we erase from x all event occurrences except those of a and b, the result differing from ε is either $baba \dots aba$, or $baba \dots ab$. For instance, the COSY program Pr_1 is a simple request system for *stopandrestart* with *errormessage* = *req*(*stopandrestart*).

The next auxiliary function is *npe*, **number of possible enablings**, and is defined as follows:

$npe : Ev \times (MFS \cup IMFS) \longrightarrow \{0, 1, \dots, \infty\}$

and where $A, A_i \in Ind, t \in Ind^*$

(1) $(\forall b \in Ev) npe(b, \varepsilon) = \begin{cases} 0 & \{b\} \notin menabled(\varepsilon) \\ 1 & \{b\} \in menabled(\varepsilon) \end{cases}$

(2) $(\forall b \in Ev)(\forall tA \in MFS) npe(b, tA)$
$= \begin{cases} npe(b,t) & \{b\} \notin menabled(tA) \\ npe(b,t)+1 & \{b\} \in menabled(tA) \end{cases}$

(3) $(\forall b \in Ev)(\forall A_1 A_2 \dots \in IMFS) npe(b, A_1 A_2 \dots) = k$
$:\Leftrightarrow (\exists j)(\forall i \geq j) npe(b, A_1 \dots A_i) = k$

(4) $(\forall b \in Ev)(\forall A_1 A_2 \dots \in IMFS) npe(b, A_1 A_2 \dots) = \infty$
$:\Leftrightarrow (\forall i)(\exists j) npe(b, A_1 \dots A_j) = i$.

The function $npe(b,t)$ describes how many times b was enabled during t.

Let Pr =< $P_1 || \dots || P_n$ be a simple request system for $a \in Ev$. We will say that Pr is **starvation free for** a if and only if:

(1) Pr is deadlock-free,

(2) $(\forall t \in IMFS) npe(req(a), t) = \infty$,

(3) $(\forall A_1 A_2 \dots \in IMFS)(req(a) \in A_j \Rightarrow (\exists k \geq 1) a \in A_{i+k})$,

where $A_1, A_2, \dots \in Ind$.

The first condition is obvious. A deadlock implies the starvation of the whole system. If Pr is deadlock-free, then its infinite behaviour unambiguously defines its finite behaviour, since in such a case MFS can be defined as the set of all finite prefixes of elements of $IMFS$. Condition (2) says that the only way of permanently blocking the event $req(a)$ from occurring is to deadlock the whole system. Otherwise $req(a)$ may be disabled only temporarily. Since $req(a)$ is not involved in any conflict (see the definition of a simple request system), if Pr is deadlock-free, nothing can prevent $req(a)$ from occurring an infinite number of times. However, because $req(a)$ belongs to one sequential component only, it might refuse to become active ("commit suicide"), without deadlocking the whole system. The last condition (3) says that any request for a is served after a certain finite amount of time. Hence, the above definition seems to meet the intuitive requirements of starvation-freeness. The above model has been used to analyze starvation problems in programs, such as the classical second problem for Readers and Writers [21][22]. Lack of space prevents us from presenting this or similar more complex examples at this point.

5 Final Comment

In this paper we have briefly indicated how to express and reason about some of the properties of concurrent systems involving priorities. The type of priority discussed is a constrained kind of *static* priority, that is, the priority constraint holding for two events will remain the same for all possible behaviours of the system involving them. In [22] we have shown how certain kinds of *dynamic* priority may be expressed and reasoned about using multiple sequence semantics and "ghost" events (compare [23]), which never occur, but serve to pass priority constraints from events to events, thus allowing the simulation of this kind of dynamic priority. Furthermore, in [14][22] we show how conventional vector firing sequence semantics can be used to reason about priority systems, provided one is not interested in such properties as starvation and atomicity, but only in such properties as serializability, deadlock-freeness, adequacy, and the preservation of capacity bounds. Conventional vector firing sequence semantics may also be used to facilitate intermediate reasoning about starvation and fairness, analogous to the type of intermediate reasoning in terms of imaginary numbers in number theory. The restrictive case when the priority relation is transitive and all aspects of behaviour are expressible in terms of partial (causal) orders is also analyzed in [24]. Finally, we have been developing a different approach to expressing priority which may prove to be more directly applicable for the expression of dynamic priorities without the use of "ghost" variables. But the current approach does suffice to prove starvation freeness and fairness of all the static priority constraints stated for COSY specifications published in the literature.

Acknowledgement
The authors would like to thank the anonymous referees for helpful comments. The first author was partially supported by a grant from the Science and Research Board of McMaster University, and the second author was partially supported by a grant from the National Science and Engineering Research Council of Canada.

References

[1] W.J. Quirk (Ed.): *Verification and Validation of Real-time Software*, Springer Verlag, (1985), 245 pp.

[2] P. E. Lauer, P. R. Torrigiani, R. Devillers: A COSY Banker, *Lecture Notes in Computer Science 83*, Springer Verlag, (1980), pp. 223-239.

[3] Inmos Ltd.: *occam:Programming Manual*, Prentice Hall, (1987), 96 pp.

[4] C.A.R. Hoare: *Communicating Sequential Processes*, Prentice Hall, (1985), 256 pp.

[5] A. W. Roscoe: Denotational Semantics for *occam*, *Lecture Notes in Computer Science 197*, Springer Verlag, (1984), pp. 306-329.

[6] L. Lamport: What it means for a concurrent program to satisfy a specification: Why no one has specified priority, *3rd ACM Symp. on Principles of Distributed Computing*, Vancouver, (1984), pp. 78-83.

[7] S. D. Brookes, C.A.R. Hoare, A.W. Roscoe: A Theory of Communicating Sequential Processes, *JACM*, (July, 1984), pp. 560-599.

[8] E. R. Olderog: Operational Petri Net Semantics for CCSP, *Lecture Notes in Computer Science 266*, Springer Verlag, (1987), pp. 196-223.

[9] E. Best: COSY:Its Relation to Nets and CSP, *Lecture Notes in Computer Science 255*, Springer Verlag, (1986), pp. 416-440.

[10] W. Reisig: *Petri Nets*, Springer Verlag, (1985), 161 pp.

[11] V. Pratt: Modelling Concurrency with Partial Orders, *Int. J. of Parallel Programming*, (January,1986), pp. 33-71.

[12] R. Milner: Calculi for Synchrony and Asynchrony, *Theoretical Computer Science 25*, (1983), pp. 267-320.

[13] M. Yoeli, T. Etzion: Behavioural Equivalence of Concurrent Systems, *Informatik-Fachberichte 66*, Springer Verlag, (1983), pp. 292-305.

[14] R. Janicki: A Formal Semantics for Concurrent Systems with a Priority Relation, *Acta Informatica 24*, (1987), pp. 33-55.

[15] P. E. Lauer, R. H. Campbell: Formal Semantics for a Class of High-level Primitives for Coordinating Concurrent Processes, *Acta Informatica 5*, (1975), pp. 297-332.

[16] P. E. Lauer, P. R. Torrigiani, M. W. Shields: COSY:A System Specification Language based on Paths and Processes, *Acta Informatica 12*, (1979), pp. 109-158.

[17] P. E. Lauer: Computer System Dossiers, in *Distributed Computer Systems : Synchronization, Control and Communication*, (Eds. E. Paker, J.-P. Verjus), Academic Press, (1983), pp. 109-147.

[18] P. E. Lauer: The COSY Approach to Distributed Computing Systems, in *Distributed Computing Systems Programme*, Peter Peregrinus, London, (1984), pp. 107-125.

[19] M. W. Shields: Adequate Path Expressions, *Lecture Notes in Computer Science 70*, Springer Verlag, (1979), pp. 249-265.

[20] W. Reisig: Partial Order Semantics versus Interleaving Semantics for CSP-like Languages and its Impact on Fairness, *Lecture Notes in Computer Science 172*, Springer Verlag, (1984), pp. 403-413.

[21] P. J. Courtois, F. Heymans, D. L. Parnas: Concurrent Control with 'Readers' and 'Writers', *Comm.ACM 14*, (1971), pp. 667-668.

[22] R. Janicki, P. E. Lauer: *The Specification and Analysis of Concurrent Systems: The COSY Approach*, Manuscript of Book to appear in 1988-1989.

[23] F. Okulicka: *The Semantics of Path Expressions with Priorities*, Ph.D. Thesis, Institute of Mathematics, Warsaw Technical University, Poland, (1986).

[24] M. W. Shields: *On the Non-sequential Behaviours of Systems Posessing a General Free-choice Property*, Report CRS-92-81, Dept. of Computer Science, University of Edinburgh, 1981.

[25] P. E. Lauer: Synchronization of Concurrent Processes without Globality Assumptions, *New Advances in Distributed Computer Systems*, (Ed. K. G. Beauchamp), Nato Advanced Study Institutes Series, D. Reidel Publishing Co., (1982), pp. 341-365.

A PETRI NET METHOD FOR THE FORMAL VERIFICATION OF PARALLEL PROCESSES

Kenneth G. Krauss
Computer Science Department
Lafayette College
Easton, PA 18042

Samuel L. Gulden
Department of CSEE
Lehigh University
Bethlehem, PA 18015

Abstract--Anyone who has seriously considered the subject realizes that the formal verification of computer programs is a complex issue. Various approaches have been taken in an attempt to find a method by which programs can be proven to be correct. The axiomatic method proposed by Hoare[1] is ideal theoretically but extremely impractical.

Robert M. Keller has proposed describing a process as a labelled Petri net.[2] This approach has certain advantages over a purely axiomatic method. We formalize and extend Keller's method. In this approach, a predicate, the truth of which establishes the program correctness, must be derived and demonstrated to be q_0-inductive.

We define a special form of a labelled Petri Net which we refer to as a process net. It is shown how a program which is constructed using accepted control structures for sequential or parallel processes can be described as a process net. We further demonstrate how we can derive a q_0-inductive predicate which establishes the program correctness.

Definitions

A place-transition graph is a system consisting of

1. Two sets P and T such that $P \cap T = \phi$. We assume P is finite and T is countable.

2. A set, E, of objects called edges.

3. Two functions, b and f that map edges to $P \cup T$ such that for any edge, $e \in E$, if $b(e) \in P$ then $f(e) \in T$ and if $b(e) \in T$ then $f(e) \in P$.

We will indicate elements of P, called places by, O's and elements of T, called transitions, by ▭'s. Given an edge, e, we call b(e) an input node of f(e) and f(e) an output node of b(e). Edges are indicated by arrows.[2]

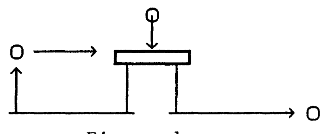


Figure 1

Assume a bipartite graph $\eta = (P,T,b,f)$ as described above. In order to refer to the places and transitions, let the places be labelled with $p_1, p_2, \ldots, p_k$ and the transitions with $t_1, t_2, \ldots, t_n, \ldots$. Also each transition, t, is labelled further with a statement of the form,

$$\text{when } B(\xi) \text{ do } \xi \leftarrow F_t(\xi)$$

where B is a predicate on ξ, a data state vector variable, and F_t is a function which assigns a value to each component of the vector ξ. In the case that $F_t(\xi) = \xi$ we reduce the transition label to, when $B(\xi)$, and in the case that $B(\xi)$ is true for all ξ we write, $\xi \leftarrow F_t(\xi)$.[2]

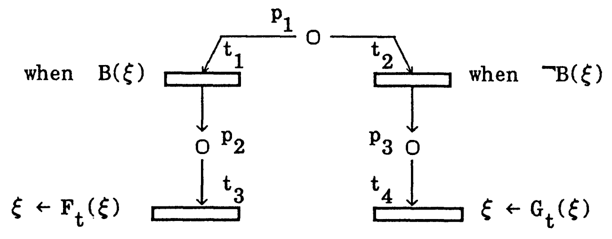


Figure 2

Let there be a control state variable consisting of a count at each place, p, designated by $\nu(p)$ where $\nu(p)$ is a non-negative integer, $(0 \leq \nu(p))$. Let N be a state of the vector obtained from all the $\nu(p)$ and ξ be a state of the data vector. Where t represents a transition node, we define (N,ξ) (N',ξ') to mean the following: ($I(t)$, $O(t)$ and $\sim$ defined after vii)

i. If $p \in I(t)$ then $\nu(p) \geq 1$.

ii. If $p \in I(t) \sim O(t)$ then $\nu'(p) = \nu(p)-1$.

iii. If $p \in O(t) \sim I(t)$ then $\nu'(p) = \nu(p)+1$.

iv. If $p \in I(t) \cap O(t)$ then $\nu'(p) = \nu(p)$.

v. If $p \in P \sim (I(t) \cup O(t)$ then $\nu'(p) = \nu(p)$.

vi. $B(\xi)$ is true.

vii. $\xi' = F_t(\xi)$.

where $N = (\nu(p_1), \nu(p_2), \ldots, \nu(p_k))$, $I(t)$ is the set of input place nodes of t, $O(t)$ is the set of output place nodes of t, B is the predicate on the data state which is associated with t, and $\sim$ is set difference. Let $Q = \{q \mid q = (N,\xi)\}$.[2]

We will refer to $(N,\xi) \xrightarrow{t} (N',\xi')$ as firing the transition t. Firing a transition moves the transition system from one state $q = (N,\xi)$ to another state $q' = (N',\xi')$ as given by i through vii above. The necessary conditions to fire a transition, t,, are given in i and vi above. If i and vi are satisfied then we say t is enabled.

The next step is to extend the binary relation, $\rightarrow$, recursively as follows:[2]

1. $\forall q(q \in Q \Rightarrow q \xrightarrow{*} q)$ where the new binary relation is denoted by $\xrightarrow{*}$.

2. $\forall q,q',q''((q \xrightarrow{*} q') \wedge (q' \longrightarrow q'') \Rightarrow (q \xrightarrow{*} q''))$ where $q,q',q'' \in Q$ and $\wedge$ means and.

We say that q' is reachable from q iff $q \xrightarrow{*} q'$.

We will denote the initial state by q_0.

Method

In order to apply a place-transition graph, with its underlying transition-system $(Q,\rightarrow)$, to the study of program correctness we consider predicates defined on Q over portions of the graph. Let P be a predicate defined on Q. For each $q \in Q$, P(q) is either true or false.

A predicate, P, defined on Q is said to be q_0-inductive iff[2]

i. $P(q_0)$.

ii. For each $q,q' \in Q$ if $q \rightarrow q'$ and P(q) then P(q').

Following [2] we consider a program correct if we can find a predicate Q with the following properties:

1. If the predicate is true then the desired conditions hold.

2. The predicate is q_0-inductive within the required portion of the graph.

We will refer to the labelled graphs that we have described as process nets. A program which uses only accepted control structures can always be modelled by a process net. We do not consider subprograms here.

Selection.

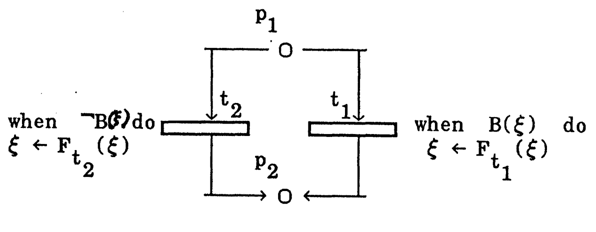


Figure 3

Iteration.

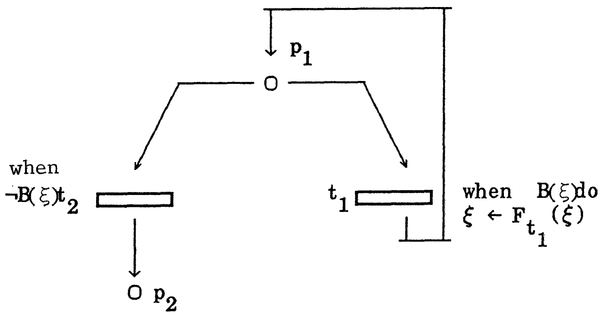


Figure 4

Parallel Processes

The issue of proving parallel programs to be correct is far more complex than proving the correctness of sequential programs. A number of statements can be executed simultaneously and the programmer may have no knowledge of the order in which parallel processes will be treated. The order can be different each time the program is executed. This situation makes testing and verification of parallel programs very difficult.

Some limitations and control must be placed on the manner in which parallel execution of statements can take place. We will extend the set of control structures by adding two more as proposed by Susan Owicki and David Gries. The two new statements are designed to implement and control parallel processing.[3]

When parallel execution is about to take place a statement of the following form is used:

resource r_1(variable list),...,r_m(variable list):

cobegin $S_1 \| \ldots \| S_n$ coend

where $S_1,\ldots,S_n$ are statements to be executed in parallel and each r_i identifies a list of variables, possibly one variable, which must be protected within the parallel execution. No variable may be common to two or more r_i.[3]

The second statement, which is to be used only within parallel execution is of the form:

with r when B do S

where r is a resource, B is a Boolean expression and S is an executable statement which does not contain a with statement referencing the same r.[3]

When we enter a parallel execution region we list all variables which can be altered or inspected from within critical sections only. Then from within the parallel execution statement the only way to access these variables is from within a critical section. That is, by using a "with r" statement.

Variables which are changed by one parallel process, S, must be in a resource if any other process that can be executed in parallel with S refers to them either for inspection or assignment. This restriction does not tell us the order in which assignments to and inspections of variables within parallel execution must be made, but it does assure us that any particular assignment or inspection will be completed before another one begins.

For the process net of the cobegin-coend statement we include a place for each variable which is in a resource. A place which indicates that the parallel execution statement is executing is also included.

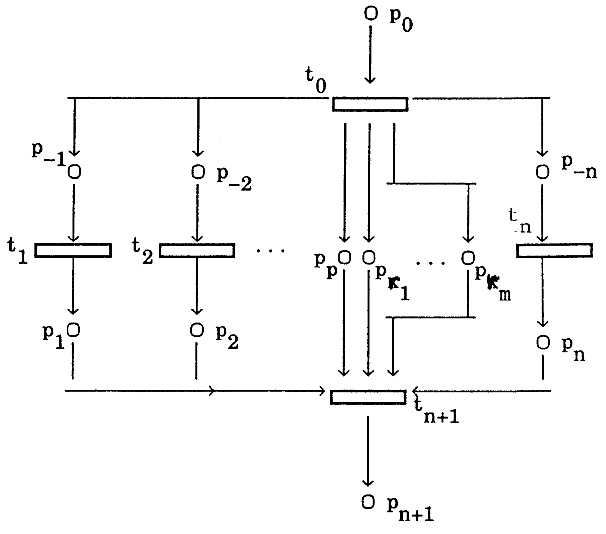


Figure 5

The with-when statement is modelled by the following process net.

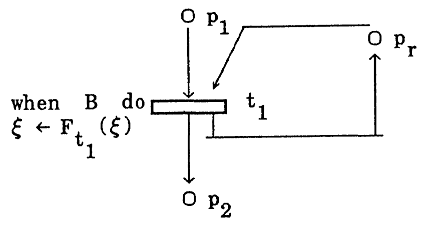


Figure 6

Notice that S is reflected by $\xi \leftarrow F_{t_1}(\xi)$. This single transition can be extended to model S if necessary. Only one process can exist in this extension at a time due to the protection provided by the place for resource r, p_r.

The method is demonstrated by the following simple example. The variable, x, is incremented by 2. In order to accomplish this the assignment x := x+1 is executed twice in parallel. Here is the program in statement form.[3]

```
begin
   resource r(x):
      cobegin
         with r when true do
            x := x+1
      || with r when true do
            x := x+1
      coend
end
```

the process net for the program is

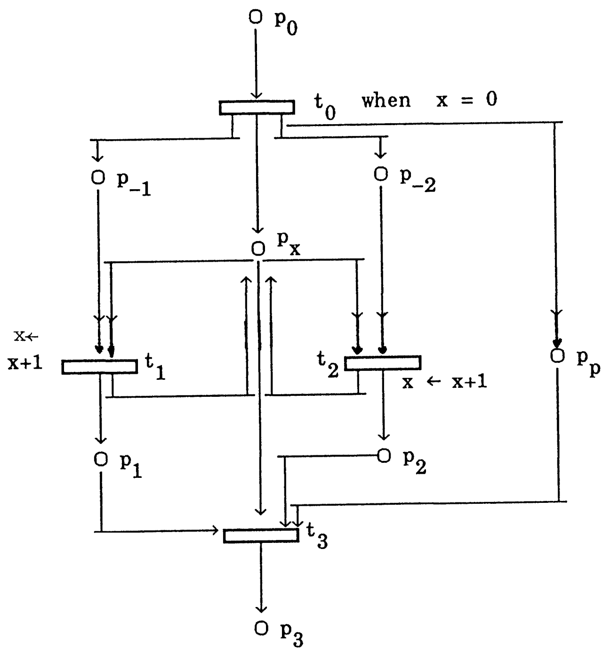


Figure 7

$$\begin{aligned}P \equiv\ &(x=\nu_1+\nu_2+2\nu_3)\wedge(\nu_0+\nu_p+\nu_3=1)\wedge\\ &(\nu_{-1}+\nu_1\leq 1)\wedge(\nu_{-2}+\nu_2\leq 1)\wedge\\ &(\nu_{-1}+\nu_1+\nu_{-2}+\nu_2>0 \Longleftrightarrow \nu_p=1)\end{aligned}$$

will serve as the q_0-inductive predicate. If $\nu_3 = 1$ then $\nu p = 0$ so $\nu_1 = \nu_2 = 0$ and $x = 2$. P is true for the initial state $q_0 = (\nu_0,\nu_{-1},\nu_1,\nu_{-2},\nu_2,\nu_x,\nu_p,\nu_3,x) = (1,0,0,0,0,0,0,0,0)$.

Assume P is true and t_0 is ready to fire. Before t_0 fires:
P, $\nu_0 = 1$, $x = 0$ which imply $\nu_{-1} = \nu_1 = \nu_{-2} = \nu_2 = \nu_p = \nu_3 = 0$

After t_0 fires:

$\nu_0' = 0$, $\nu_x' > 0$, $\nu_{-1}' = \nu_{-2}' = \nu_p' = 1$, $\nu_3' = \nu_1' = \nu_2' = 0$, $x' = 0$.

Hence P is true after t_0 fires.

Assume P is true and t_1 is ready to fire. Before t_1 fires:

$P, \nu_{-1} = 1, \nu_x > 0$ imply $\nu_p = 1, \nu_0 = \nu_3 = \nu_1 = 0$, $x = \nu_2$.

After t_1 fires:

$\nu'_{-1} = \nu'_0 = \nu'_3 = 0, \nu'_1 = \nu'_\rho = 1, \nu'_2 = \nu_2, x' = x+1 = \nu'_1 + \nu'_2 + 2\nu'_3, \nu'_{-2} = \nu_{-2}$.

Hence P is true after t_1 fires.

The same is true for t_2 with a similar argument.

Assume P is true and t_3 is ready to fire. Before t_3 fires:

$P, \nu_1 = \nu_2 = \nu_p = 1, \nu_x > 0$ imply $\nu_0 = \nu_3 = \nu_{-1} = \nu_{-2} = 0, x = 2$

After t_3 fires:

$\nu'_1 = \nu'_2 = \nu'_p = \nu'_{1} = \nu'_{2} = \nu' = 0, \nu'_3 = 1, x' = x = 2$.

Hence P is true after t_3 fires.

It follows that the program is correct.

Mutual Exclusion

G. L. Peterson developed a solution to the two-process mutual exclusion problem.[4] The protocol for process one is given as follows:

```
Q1:= true;
TURN:= 1;
wait until not Q2 or TURN =2;
Critical Section;
Q1:= false.
```

We give here the process net for process one which is half of the process net for the parallel execution of process one and process two.

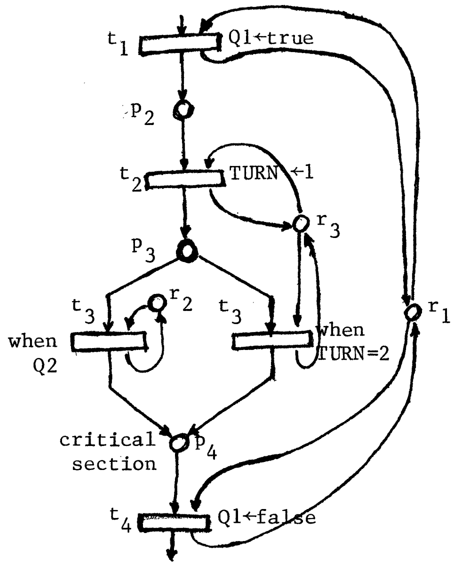


Figure 8

In order to prove that Peterson's solution is correct we can establish the following:

1) Deadlock can not occur. That is to say process 1 can enter its critical region if Process 2 is not trying and vice versa.

P1: $(\nu'_2+\nu'_3+\nu'_4=0 \Leftrightarrow \neg Q2) \wedge (\nu_2+\nu_3+\nu_4=0 \Leftrightarrow \neg Q1)$. Also one process can enter if both are trying to enter their respective critical regions.

P2: $(\nu_3+\nu'_3=2 \Rightarrow (TURN=1 \vee TURN=2))$

2) Mutual Exclusion is maintained.

$(\nu_4=1 \wedge \nu'_4=1) \Rightarrow$ false.

It can also be shown that indefinite postponement can not occur with this solution.

In order to show that $(\nu_4=1 \wedge \nu'_4=1) \Rightarrow$ false is q_o-inductive we consider

P3: $(\nu_4=1) \Rightarrow ((\neg Q2 \vee TURN=2) \wedge Q1)$ and

P4: $(\nu'_4=1) \Rightarrow ((\neg Q1 \vee TURN=1) \wedge Q2)$ are q_o-inductive. Notice that $(\nu_4=1) \wedge (\nu'=1)$ implies the contradiction $((\neg Q2 \vee TURN=2) \wedge Q1) \wedge ((\neg Q1 \vee TURN=1) \wedge Q2)$.

It follows from the discussion above that the predicate that we must extablish as being q_o-inductive throughout the process net is:

$P \equiv P1 \wedge P2 \wedge P3 \wedge P4 \wedge \sum_{i=1}^{6} \nu_i \leq 1 \wedge \sum_{i=1}^{6} \nu'_i \leq 1$.

Let $q_o = (\nu_0, \nu_1, \nu_2, \nu_3, \nu_4, \nu_5, \nu_6, \nu'_1, \nu'_2, \nu'_3, \nu'_4, \nu'_5, \nu'_6, \nu_7, \nu_p, \nu_{Q1}, \nu_{Q2}, TURN, Q1, Q2, TURN) =$

(1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,

false,false,1)

With the process net, predicate, and q_o we can establish the correctness of Peterson's solution.

Bibliography

[1] Hoare, C. A. R.,. An Axiomatic Basis for Computer Programming, Communications of the ACM, Vol. 19, Number 7, July 1976.

[2] Keller, R. M., Formal Verification of Parallel Programs, Communications of the ACM, Vol. 19, Number 7, July 1976.

[3] Owicki, Gries, Verifying Properties of Parallel Programs: An Axiomatic Approach, Communications of the ACM, May 1976, Vol. 19, Number 5.

[4] Peterson, G. L., Myths About the Mutual Exclusion Problem, Information Processing Letters, Vol. 12, Number 3, March 1981, North-Holland.

Dynamic Scheduling and Memory Management for Parallel Programs

Michael Weiss, C. Robert Morgan, and Peter Belmont
Compass, Inc.*
Wakefield, MA

Zhixi Fang
Concurrent Computer Corporation
Tinton Falls, NJ

Abstract

We describe a scheme for the dynamic scheduling of DOALL-loops and (a generalization of) FORK-JOINs. While our runtime model is generally applicable, we are especially concerned with supporting the output of a parallelizing FORTRAN compiler for a shared memory multiprocessor architecture. We assume that the user wants to explicitly code parallelism. We build on the work of Fang et al. [5]. Our scheme is two-level, like theirs, and permits general nested loops. We use a queue for the upper level, however, instead of a complex system of tables.

Memory management issues are also addressed. Efficiency concerns dictate treating process locality differently from procedure locality. As a corollary, we have the following interesting result: the number of call frames required at any one time for a nonrecursive procedure is bounded by the number of processors.

1 Introduction

The wide variety of MIMD supercomputing systems has created a demand for software techniques to help exploit the new processing power. Much effort has been devoted to developing restructuring compilers and source-to-source translators that detect implicit parallelism in a "dusty deck" program. Such automatic parallelization is not always sufficient— the user often needs to code in parallelism. DOALL-loops and FORK-JOINs are common parallel constructs.

The runtime support for the new parallel constructs will have a major impact on the success of a parallel compiler; scheduling must be done with an intelligent trade-off of overhead against concurrency. Since the user can code the parallel constructs explicity, the runtime model should restrict their use as little as possible. For example, parallel loops should be nestable with each other, with other parallel constructs, and with all sequential constructs, including arbitrary control flow.

A macro package interacting with the operating system provides the simplest way to utilize multiple processors from within a single FORTRAN program (see, e.g., Dongarra and Sorensen [4]). However, the overhead to start an operating system task is usually high and geared to general purpose use rather than the particular needs of the parallel program. A better approach is to build a "mini-OS" within the existing operating system. We will refer to the units scheduled by the mini-OS as *processes*, in contrast to the *tasks* of the surrounding general purpose OS.

The operating system analogy suggests using a queue for scheduling processes. On the other hand, unnested parallel loops are usually scheduled using a pair of counters. Fang et al. have described one scheduling strategy for general nested loops [5]. (To our knowledge, this is the only previous work that deals with the general nested case.) Somewhat surprisingly, their approach is not queue-based. They schedule innermost parallel loops using a pair of counters ("low level scheduling"), but use a complex system of tables for outer parallel loops ("upper level scheduling"). The compiler constructs the tables from dependence information. The main drawback to their scheme is the size of the tables. It is also not clear how to incorporate parallel procedure calls into their system.

In Section 3, we describe a two-level scheduling scheme where the upper level is queue-based. This scheme handles the general nested case and is both simpler and more efficient than the table-based approach; it integrates smoothly with procedure calls.

Memory management can also make or break the runtime system. Much previous work has testified to the feasibility and desirability of "directly parallelizing call statements" [2,9] [3, Chapter 4]. Allowing nested parallelism leads to a *process tree*. Incorporating procedure calls leads to a picture like Figure 1.

Separate procedure invocations of course require separate frames for local variables. A cactus stack implementation is inevitable. However, different processes within the same procedure also have locality needs. Interaction with the memory management system every time processes are spawned would incur prohibitive overhead.

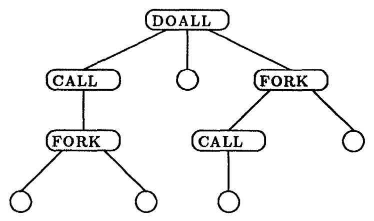


Figure 1: A Process Tree

In Section 4, we describe a method for handling procedure locality using a cactus stack (i.e., dynamically), but process locality statically. The overhead with this scheme should be little more than the normal overhead of procedure calls. An interesting theoretical consequence is a bound on the number of frames that a nonrecursive procedure can simultaneously require.

2 Parallel Constructs in the Model

Our runtime model is designed to support DOALL-loops and a generalization of FORK-JOINs, which we call a *FORK-JOIN DAG*. We allow arbitrary nesting of the these constructs along with ordinary sequential control-flow constructs such as DO-loops. Our model also permits procedure calls inside of parallel constructs, where the called procedure may itself contain parallelism.

* Formerly Massachusetts Computer Associates, Inc.

By a DOALL-loop, we mean the most straightforward (asynchronous) parallel version of the conventional DO-loop: a process entering the DOALL-loop splits into subprocesses, one for each iteration, and these subprocesses merge back together at the end of the loop. What order the iterations are actually performed in depends on the assignment of processors to processes; we make no assumptions about this. Note that the processor entering the DOALL-loop is not necessarily the same processor that executes the code following the loop.

FORK-JOINs are another familiar parallel construct. We permit a generalization we call a FORK-JOIN DAG: that is, a single-entry single-exit DAG of FORKs and JOINs. More precisely, each node in the DAG is FORK, a JOIN, or represents a construct other than a DAG[1].

FORK-JOIN DAGs appear quite naturally in the output of a parallelizing compiler that exploits coarse-grain concurrency. Outside of loops (or restricting attention to a single iteration of a loop), the data dependence graph is a DAG. This naturally translates into a FORK-JOIN DAG. If a node in the dependence graph has several predecessors, that implies a JOIN just before it. If a node has several successors, that implies a FORK just after it. An entry FORK and an exit JOIN are added if necessary to ensure a single-entry single-exit construct.

In practice, the compiler would not directly translate the dependence graph into a FORK-JOIN DAG in this fashion, but would first "collapse" the dependence graph into a smaller DAG to achieve a good trade-off between the overhead of the FORKs and JOINs, and the expected speed-up due to parallelism. We will discuss one such collapsing algorithm in a subsequent paper [10]; see also Sarkar and Hennessy [7]. In any case, requiring the FORK-JOIN DAG to reduce to a combination of FORK-JOINs places an unnecessary burden on the compiler's output. We refer to the JOINs in a FORK-JOIN DAG, and also to the end of a DOALL-loop, as *merge points.*

3 Scheduling

We will describe our scheduling strategy as an optimized version of a conceptually simple (but inefficient) method that we call the *idealized scheme.* The next section outlines the idealized scheme; subsequent sections describe several optimizations that make it practical.

Our DOALL loops will "count down," that is, be scheduled from higher index values to lower index values. This allows loop termination to be detected by a "test for zero," which is more convenient than a test against the upper bound.

3.1 The Idealized Scheme

The essential features of the idealized scheme are listed below.

- There is a list of *process descriptors.* A descriptor contains the information needed to create a process. This includes its starting address, the base address of local variables, and iteration indices for all enclosing parallel loops.
- The program uses a fixed set of (operating system) tasks, called *drivers.* All drivers are identical; they share the same code and data segments. The *dispatch loop* assigns drivers to processes. Whenever a driver becomes idle, it jumps to the dispatch loop, where it tries to get a process from the descriptor list. If it succeeds, it removes the descriptor from the list; if it fails, it stays in the dispatch loop.
- Whenever a process splits into several processes, the driver executing the process picks one of the child processes to execute and places descriptors for the other child processes on the descriptor list. For example, a process executing "`DOALL I=1,10`" splits into 10 processes; its executing driver might choose to execute the "`I=10`" process itself, placing descriptors for the other 9 processes on the list.
- The synchronization implied at a merge point is obtained using *merge counters.* Each driver entering the merge point decrements the counter. The last driver to enter continues with the code past the merge point; the others jump to the dispatch loop.
- *Process management code* is code other than the dispatch loop that is needed to handle the spawning and merging of processes. This consists of forking code for FORKs and DOALLs (which places descriptors in the list), and merging code for merge points (which decrements the merge counter and tests to see if it's zero). Thus FORKs, JOINs, DOALLs, and ENDALLs correspond to distinct chunks of code in the object code module.

3.2 Innermost Loops

If L is an innermost DOALL-loop (i.e., there are no parallel constructs nested inside L), then the idealized model will most likely cause a driver to get descriptors repeatedly from the list for different iterations of L. Since nothing is changing except the iteration index, it makes sense to keep the driver assigned to L until L is exhausted, without revisiting the dispatch loop.

We accomplish this as follows:

- A shared global variable (call it A) holds the highest unassigned value of the iteration index. A driver that is assigned to L gets a new iteration by performing a **`fetch&decrement`** operation on A.
- There is a single descriptor on the descriptor list to assign drivers to L in the first place. The first driver to encounter the DOALL statement for L enqueues this descriptor. When a driver is assigned the last iteration in L, it removes the descriptor.
- A driver assigned to L gets all the information it needs to execute L from this descriptor, other than the iteration index (e.g., iteration indices for enclosing DOALL-loops).
- The merge counter is another shared global variable controlling execution of L; we call it B, and refer to A and B together as the *AB-pair* for the innermost loop.

Many machines have an indivisible **`fetch&decrement`** operation, so that locking and unlocking the descriptor list is not necessary.

We have as yet said nothing about the allocation of AB-pairs. One obvious approach is *dynamic allocation*: keep a pool of slots for AB-pairs; the driver that enqueues the descriptor gets a free slot from the pool and passes it in the descriptor; the driver that deletes the descriptor deallocates the slot. However, we argue in Section 4 that *static allocation* (described there) is more efficient.

If the number of processors is sufficiently large and the upper bound of the DOALL-loop is also large, the AB-pair could become a hot spot. This can be dealt with by means of the software combining tree of Yew et al. [11][2], or by means of Brooks' "butterfly barrier" [1].

[1]E.g., an assignment statement, a DO-loop, or a DOALL-loop. There is no point in nesting a DAG inside a DAG, since the result is equivalent to one larger DAG.

[2]But note that Rettberg and Thomas report that on one system (the BBN butterfly), simple **`fetch&add`** outperforms software combination for less than about 100 processors [6]. We would expect the crossover point to be lower for a bus-based system.

3.3 Combining descriptors

Consider the following nested DOALL loops:

```
DOALL I=1,3
 DOALL J=1,2
  X(I,J)
 ENDALL
 DOALL J=1,2
  Y(I,J)
 ENDALL
ENDALL
```

Figure 2 illustrates how one process splits into three processes, each of these splitting again. Consider a driver that flows down the rightmost branches, eventually executing the box $\boxed{X_{32}}$. Even with the innermost loop optimization, the idealized model would put up descriptors for "I=2" and "I=1", plus a descriptor pointing to the AB-pair for the first "DOALL J" loop.

We can combine these into one descriptor. This descriptor has "I=3" in it, and represents the fact that there is unassigned work for this and lower values of "I". When a driver is assigned to the $\boxed{X_{12}}$ box, it decrements the "I" in the descriptor. Eventually a driver is assigned to the $\boxed{X_{11}}$ box, at which point it removes the descriptor.

In general, this combining of descriptors can occur only when a DOALL-loop occurs as the first executable part of an enclosing DOALL-loop. Three different descriptors will be needed for second "DOALL J" loop, carrying the information "I=3", "I=2", and "I=1".

Figure 2 shows a snapshot of the relationship between the descriptor queue and the AB-pairs for the above example. For clarity, we have shown the queue as if AB-pairs were dynamically allocated. With static allocation, the value of "I" and of "start" are enough to determine the address of the AB-pair, as will become clear in Section 4.

If there were a non-parallel statement right before the first "DOALL J" loop (say "K(I)=1"), the driver taking the "I=3" iteration would put a single descriptor up with "I=2". This represents the unassigned work with "I=2" and "I=1". The driver dispatching off this descriptor would decrement "I", and the descriptor would be removed when "I" was decremented to 0.

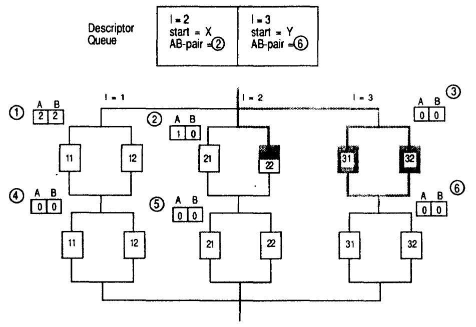


Figure 2: AB-Pairs and the Descriptor Queue

Analogous to combining descriptors, one may consider combining merge counters. Still using Figure 2 as an example, the second inner DOALL-loop requires three merge counters, each with a count of 2, and the outer DOALL-loop requires a merge counter with a count of 3. All four merge counters could be combined into a single merge counter with a count of 6. In general, when a DOALL-loop occurs as the last executable part of an enclosing DOALL-loop, the merge counters can be combined.

However, the instruction saving (one **fetch&decrement**) is minor, compared to the saving associated with combining descriptors. The four merge counters thus combined form a natural "software combining tree" [11]. We are thus increasing the likelihood that the combined merge counter will prove a hot spot.

3.4 Strip mining

Strip mining, or *blocking loops*, is a familiar transformation from vectorization; it serves a dual role in scheduling and memory management.

In our situation, we rewrite a DOALL loop as a DO loop nested inside a DOALL loop. Suppose K is a constant that the compiler chooses; usually it would be of the same order of magnitude as the number of processors. The bound expressions are chosen so that the DOALL-loop will have K iterations. (If we were vectorizing, we would arrange matters so the inner loop had K iterations.)

From a scheduling standpoint, strip mining serves to group iterations together so as to reduce spawning overhead. By making K larger than the number of processors, we can expect some load balancing to help performance.

However, the main reason for strip mining comes from memory management, as we explain in Section 4.

3.5 FORK-JOIN DAGs

For a general FORK-JOIN DAG, there is no natural way to pair up fork points with merge points. This means that the merge counters for each JOIN must be initialized by the entry FORK.

Suppose we have a FORK A which has a JOIN node B as a successor. The idealized model would spawn a process (i.e., place a descriptor on the list) solely to participate in the merge (i.e., decrement the merge counter). It is more efficient for the processor executing the FORK to directly decrement the merge counter. In general, then, a processor executing a FORK

- decrements the merge counters of any successors that are JOINs;
- makes a list of successors that are ready to be scheduled; this consists of all non-JOIN successors, plus all JOIN successors whose merge counter went to zero on the previous step;
- picks one of the successors on this list to jump to and puts up descriptors for the others.

With the above optimizations, the descriptor list must be implemented as a queue, not a stack. This points up the greater regularity of the idealized model, where these details are irrelevant. With a descriptor stack it is possible to construct a scenario with just two processors where it becomes necessary to remove a descriptor in the middle of the stack.

4 Memory Management

Concurrency forces replacement of the traditional stack of call frames (or even static allocation) with a tree-like structure known as a *cactus stack*. The cactus stack deals with the local/global (or private/shared) issues implied by having parallelism and procedure calls together.

One simple way to define the cactus stack is to have it directly mirror the process tree. That is, when a process with a given frame splits into several processes, we create a child frame for each child process. The only other way child frames are created is through the usual procedure calling mechanism.

While appealing in its simplicity, this scheme calls for frequent allocation and deallocation of frames, causing excessive runtime overhead. We propose another approach, which treats *process locality* differently from *procedure locality*. We will allocate a new frame only on a procedure call. Each frame is labelled with the name of the procedure whose invocation created it. Initially, a frame has just one process active in it. If that process splits, then the child processes share the same frame. If two child processes make procedure calls, they are allocated different child frames.

Procedure locality is thus handled by means of the cactus stack. Process locality can be handled by the well-known technique of *scalar promotion* (also called *scalar expansion*). Suppose a DOALL-loop "`DOALL I=1,100`" contains a variable V for which each iteration (i.e., subprocess) requires a separate copy. The compiler will scalar promote V to an array $V(1:100)$.

Two objections can be answered immediately.

- Strip mining bounds the size of the promoted arrays, if the bounds of the DOALL-loop are unknown at compile time, or even known but large.
- A processor usually has some local memory, perhaps some registers, even with a shared memory architecture. Using these for process-local variables is preferable to allocating arrays.

  However, we cannot simply assume that all process-local variables will be stored in registers, and not only because of the danger of running out of registers. If a process splits into subprocesses that later merge back together, any process-local variable whose lifetime spans the split cannot be allocated in a register. This follows from the observation that the processor continuing after the merge need not be the same processor that entered the splitting point.

  We suggest that these trade-offs are best handled by standard register allocation techniques. The register allocator must treat a fork point or a merge point as a boundary that kills all register assignments. (It may still help the register allocator to mark promoted variables as different from ordinary array references.)

What variables are process-local? Most notably, merge counters and AB-pairs. That is, if a "`DOALL I`" loop has another parallel construct nested inside it, the merge counter for the inner construct will require scalar promotion to an array indexed by "`I`".

The scalar promotion approach amounts *static allocation* in the sense that all process-local variables are allocated on procedure entry— no special action needs to be taken for process start-up. The time overhead for static allocation is just the time to allocate the call frame. We contrast this with *dynamic allocation* (as outlined for AB-pairs in section 3), where we look for storage for all process-local variables everytime we spawn a process. In a program with a significant amount of parallelism, we expect process creation to be much more frequent than procedure calls. Thus static allocation is generally preferable to dynamic allocation.

The essential features of this scheme are:

- The base address of a process (passed in the descriptor) points to the one frame it can access.
- If new processes are forked within a procedure, they initially share the same frame.
- A new frame is created when a process calls a procedure. As part of normal procedure calling conventions, the old base address is stored in the new frame (effectively providing a parent link in the cactus stack), and the base address of the process is changed to point to the new frame.
- The scalar promotion mechanism handles the allocation needs of process-local variables.
- When a process executes a RETURN, the following always holds: *all processes forked since procedure entry have merged.* Hence the returning process is the only one using the frame, which can be deleted.

We consider finally the problem of allocating and deallocating frames without hopelessly fragmenting memory. Perhaps the simplest approach is to adopt a fixed minimum size for call frames, and keep a free list. Procedures with modest storage needs would of course waste some space (so-called "internal fragmentation").

Procedures with large storage needs could be handled with overflow frames. Alternately, a special free list of "extra-large" frames could be set aside for them; the amount of storage required for this special free list can be bounded in most cases by the following observation:

Theorem: If P is nonrecursive (i.e., P cannot call itself directly or indirectly) then the cactus stack can contain at most n frames labelled P at any time, where n is the number of processors.

Proof: Each processor "lives" in one of the frames on the cactus stack, namely the one for the invocation of the procedure it is currently executing. Any subtree of the cactus stack has at least one processor living in it. To see that this is true, note that a processor leaves a frame in three ways: by a procedure call, in which event the processor moves to a child frame; by a procedure return, in which event the frame is deleted; and by being not-the-last to reach a merge point, in which event another processor is still active inside the parallel construct, and hence living in the subtree.

Say $P_1, \ldots, P_k$ are all the frames labelled P. The subtrees rooted at distinct P_i and P_j must be disjoint by the hypothesis that P is nonrecursive. Since each such subtree has a processor living in it, $k \leq n$. QED

5 Conclusions

We have replaced the complex tables for upper level scheduling as described by Fang et al. [5] with a simple queue-based scheme. Both schemes support general parallel nested loops. Despite the different viewpoints, most of the code for the table based approach can be readily adapted to provide process management code for our approach. We reserve the details for a future paper [10].

We have also proposed using dynamic allocation (in the form of a cactus stack) for procedure calls, combined with static allocation (in the form of scalar promotion) for process-local variables. With this approach, process creation entails no time overhead for memory allocation. The overhead for procedure calls is not much greater than for a tradition call stack. We obtained also an interesting theoretical bound on the number of simultaneously active call frames for a nonrecursive procedure.

References

[1] Eugene D. Brooks III. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, August 1986.

[2] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*, July 1986.

[3] David Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, Houston, Texas, February 1987.

[4] Jack J. Dongarra and Danny C. Sorensen. *A Portable Environment for Developing Parallel Fortran Programs.* Technical Report Technical Memorandum No. 79, Argonne National Laboratory, July 1986.

[5] Zhixi Fang, Pen-Chung Yew, Peiyi Tang, and Chuan-Qi Zhu. Dynamic processor self-scheduling for general parallel nested loops. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 1–10, August 1987.

[6] Randall Rettberg and Robert Thomas. Contention is no obstacle to shared-memory multiprocessing. *Communications of the ACM*, 29(12):1202–1212, December 1986.

[7] Vivek Sarkar and John Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*, pages 17–26, 1986.

[8] Peiyi Tang, Pen-Chung Yew, Zhixi Fang, and Chuan-Qi Zhu. Deadlock prevention in processor self-scheduling for parallel nested loops. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 11–18, August 1987.

[9] Rémi Triolet, François Irigoin, and Paul Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*, pages 176–185, July 1986.

[10] Michael Weiss and Zhixi Fang. Adding parallelization and vectorization to an existing compiler. In preparation.

[11] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 51–58, IEEE, August 1986.

Efficient Dynamic Scheduling of Medium-Grained Tasks for General Purposing Parallel Processing

Albert J. Musciano
Thomas L. Sterling
Advanced Technology Department
Harris Corporation
PO Box 37, MS 3A/1912
Melbourne, FL 32902

Abstract

SpoC is a research initiative established to realize a general parallel execution environment on an existing multiprocessor. Its purpose is to investigate the architectural requirements for advanced high performance parallel architectures. SpoC uses a dynamically scheduled runtime system to execute application programs written in Simultaneous Pascal on the Concert Multiprocessor. Effective execution of medium-grained parallel tasks is achieved by implementing scheduling and synchronization primitives in software. The methods employed by the SpoC execution environment are described and their implications for advanced parallel processor architectures are discussed.

1. Introduction

A myriad of challenges confront general purpose parallel processing, from its definition to its realization. The SpoC parallel execution environment has been developed as a general purpose parallel computing research vehicle to explore these challenges. Emphasis has been placed upon minimization of task management overhead to provide good scalability, coupled with the study of advanced architectures and novel mechanisms to define the next generation of parallel computers. The result is a general purpose parallel computing system hosted by a 64 processor shared memory multiprocessor that performs efficient dynamic scheduling of medium-grained tasks defined by programs written in a high level parallel programming language. The experience gained from SpoC has provided insight into the principal elements of a more advanced parallel architecture.

The SpoC system has evolved from a minimally functional parallel programming environment[1] into an efficient parallel research tool. SpoC maps a high level virtual parallel machine, defined by the Simultaneous Pascal[2] programming language, onto the physical hardware of the Concert Multiprocessor[3]. In order to achieve a usable system, SpoC must minimize the overhead associated with dynamic scheduling. This overhead includes all the work performed to manage parallel activities that would not be required by a sequential machine. If the overhead required to schedule a task exceeds the actual work performed by that task, the scalability of the application will suffer. Scalability can be enhanced by finding a computing model whose constructs can be efficiently implemented.

Simultaneous Pascal uses a dynamically scheduled concurrent thread version of the parallel control flow model[4]. This model provides for sequential, atomic threads which execute on a processor without interruption or suspension. The termination of one or more threads may cause other threads to be created and executed. The explicit parallel constructs in Simultaneous Pascal allow thread creation and synchronization points to be determined at compile time. Armed with this knowledge, the SpoC runtime software uses a simple set of data structures to manage thread creation, execution, and synchronization. The overhead involved in these software routines yields a useful granularity of about 140 machine instructions per thread, which is within the bounds of medium-grained parallel processing. In addition, the block structured nature of Simultaneous Pascal allows SpoC to exploit the hierarchical memory of the Concert Multiprocessor, maximizing references to local memory while minimizing global references and their associated communication traffic.

Management of threads and their associated control structures can result in appreciable overhead beyond that required of a uniprocessor system. There is substantial opportunity through architectural improvement to greatly reduce this overhead. Current processors and memory controllers were not designed with parallel processing in mind, and lack the features needed to support dynamic parallel processing. The incorporation of special memory controllers along with caches and prefetching techniques into future machines will reduce the work required of the runtime software to support a dynamically scheduled parallel execution environment.

The SpoC experience provides a working model of all of the integrated functions comprising a dynamically scheduled parallel processing environment. Studies of application programs run on SpoC return profiles of the use of the underlying mechanisms. The insight derived from this activity is leading to a determination of the most likely mechanisms whose support in hardware will have the greatest impact on system performance.

2. Parallel Computing Environment

A complete parallel computing environment consists of three principal components: a medium for expressing parallel algorithms, parallel computing hardware to execute those algorithms, and a set of functions which map the parallel virtual machine to the physical parallel hardware. At Harris, these three requirements are met by the Simultaneous Pascal programming language, the Concert Multiprocessor Testbed, and the SpoC execution environment.

2.1. Simultaneous Pascal

Simultaneous Pascal is a superset of ISO Level 0 Standard Pascal[5]. It extends the sequential programming model of Pascal with parallel language constructs which support a concurrent thread version of the parallel control flow model of computation. Programmers explicitly delineate the parallel threads within their program using these constructs. A thread is a sequential piece of code which, once started, will not block or suspend until it terminates. Upon termination, the thread causes other threads to start executing. The programmer indicates the precedence relationships of the threads in his program using the various parallel language constructs in Simultaneous Pascal.

Within Simultaneous Pascal, there are three principal parallel constructs, illustrated in Figure 1. The **forall** statement provides SIMD[6] parallelism, allowing multiple instantiations of a single statement (or a block of statements) to be executed in parallel. Each instantiation is denoted by a different value of an indexing variable. The **fork** statement allows the programmer to execute several different statements in parallel, providing MIMD parallelism. Finally, the **traverse** statement allows a dynamically created data structure, such as a tree or linked list, to be operated upon in parallel. This dynamic form of SIMD parallelism instantiates a thread for each node in the data structure.

In addition to this statement level parallelism, Simultaneous Pascal provides for the parallel evaluation of complex expressions. Since expression evaluation order in Pascal is explicitly undefined, the programmer may indicate (with appropriate operator symbols) that subexpressions should be evaluated in parallel. As subexpressions are evaluated, pending operations are completed, much like the functional model of parallel computing[7].

All of the parallel constructs in Simultaneous Pascal utilize barrier synchronization to coordinate the execution of threads. When a **forall**, **fork**, **traverse**, or parallel expression is executed, all of the threads within the construct must finish executing before control will pass to the statement following the parallel construct. This synchronization of terminating threads is called a *join operation.*

Simultaneous Pascal also provides additional parallel support statements, including the **locking** statement and the **using** statement. The **locking** statement allows the programmer to define, lock, and release semaphores, providing exclusive access to shared data objects. The **using** statement extends variable scoping down to the statement level, allowing storage for variables to be allocated and released as individual statements are executed.

```
fork
   magnitude := sqrt(sqr(x) + sqr(y));
   initialize;
   for i := 1 to max do
      evaluate(i)
join

forall i := min_x to max_x do
   forall j := min_y to max_y do
      digitally_thin(image[x, y]);

type  tree_ptr  = ^tree_node;
      tree_node = record
                       data  : integer;
                       left  : tree_ptr;
                       right : tree_ptr
                  end;

var head : tree_ptr;

traverse p := head via left, right do
   evaluate(p^.data);
```

Figure 1. Simultaneous Pascal parallel statements. The **fork** statement causes the three statements (including the entire **for** statement) to execute in parallel. The nested **forall** illustrate element-wise parallelism across a two-dimensional array. Using the indicated types and variables, the **traverse** statement will access in parallel every node of the tree accessible via head.

Simultaneous Pascal specifically precludes any reference to the underlying physical parallel hardware. Programmers are unaware of the number of processors the machine will have, as well as how those processors are connected. Simultaneous Pascal encourages the programmer to think about his algorithm in terms of parallel threads, which are mapped by the compiler and runtime software onto a particular machine configuration. This abstraction of a parallel virtual machine significantly enhances the writing of easily understood, portable parallel code.

2.2. The Concert Multiprocessor Testbed

The Concert Multiprocessor Testbed is a hierarchical shared memory multiprocessor, incorporating 64 processing elements. Each processing element is comprised of a MC68000 microprocessor coupled with a MC68881 floating point unit and at least 512 Kbytes of high speed local memory. Eight processing elements are arranged into a cluster, with up to four megabytes of memory shared among the members of the cluster. Within the cluster, the processing elements and the shared memory are connected by a Multibus backplane. Contention for the shared memory is managed by a parallel arbiter employing a round robin scheduling mechanism.

Eight clusters are connected via a crossbar switch to eight megabytes of global memory. The global memory is divided into 512 Kbyte blocks, providing 16-way interleaving on word boundaries. Each cluster has an interface

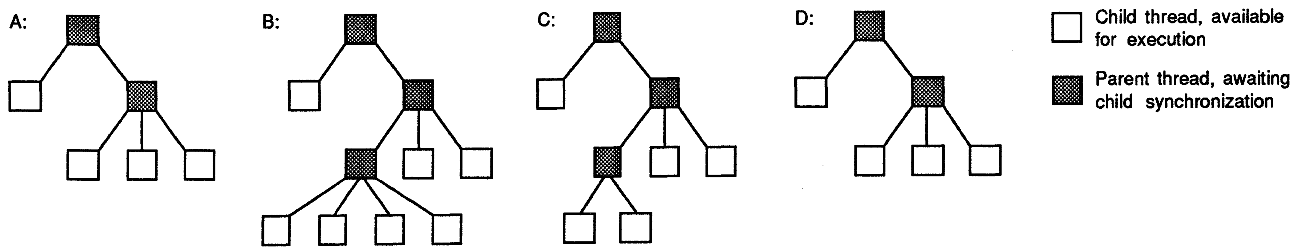

A:
B:
C:
D:
Child thread, available for execution
Parent thread, awaiting child synchronization

Figure 2. Growth and collapse of a parallelism tree. In A, an existing parallelism tree, with two parent threads, and four executing children. In B, a child thread creates four children of its own, and waits for their termination. In C, two of the four children created in B remain. Finally, in D, all four children have terminated, and the parent can resume execution.

card within it which connects to the crossbar switch. A processor within a cluster desiring global memory access first acquires access to the global memory interface controller via the Multibus. A memory request is then sent to the crossbar switch, which in turn arbitrates for access to the desired bank of global memory.

2.3. $\mathcal{S}$poC

$\mathcal{S}$poC (Simultaneous Pascal on Concert) is the synthesis of the Simultaneous Pascal programming language with the Concert hardware. The Simultaneous Pascal cross compiler generates MC68000 object code, interspersed with calls to runtime support software which implement the various Simultaneous Pascal parallel constructs. This runtime support software is the heart of the $\mathcal{S}$poC system, and provides the mapping from the virtual Simultaneous Pascal parallel machine to the physical Concert Multiprocessor. The design and implementation of this runtime software determines the efficiency of the $\mathcal{S}$poC system as a whole.

3. Representation of Work

An application program can be viewed as an amount of work to be performed. In $\mathcal{S}$poC, the programmer uses Simultaneous Pascal to describe the work, and how that work can be broken into threads. The Simultaneous Pascal compiler translates this high level representation of work and thread relationships into a lower level description which is readily executable on the Concert Multiprocessor. The manner in which the application work is represented significantly affects how well the application performs when executed on Concert.

Work in $\mathcal{S}$poC is represented by *thread objects*. Initially, a single thread object, representing the starting code of the application, is placed into the system. The idle processors contend for this object, and the processor which acquires it begins executing the application code. At some point, additional thread objects will be created. The remaining processors then acquire and execute these threads. As the system proceeds through an application, there is a constant ebb and flow of thread objects being created, waiting for an available processor, being acquired by a processor, executed, and being destroyed. Finally, the system collapses down to a single thread, which terminates the application.

3.1. Management of Parallelism

The creation and synchronization of threads occurs in a hierarchical, tree-like manner. When an executing thread encounters a parallel statement in an application, a number of child thread objects are created to represent the instantiations of the body of the parallel construct. These children retain a pointer to their parent object, and utilize a data structure in the parent to synchronize the termination of the child threads. When all of the children have terminated, the parent thread resumes execution at the point following the parallel construct. Thus, as parallel constructs are encountered, the tree grows downward, representing nested parallel statements. As these statements synchronize and terminate, the tree is reduced, communicating the synchronization upwards to the parent threads.

At any point during execution, the leaf nodes of the parallelism tree represent child threads either executing or awaiting execution. The interior nodes of the tree represent parent threads, awaiting the termination of their children. As the children terminate, the interior nodes are re-exposed as leaf nodes, and resume execution. The growth and collapse of a parallelism tree is shown in Figure 2.

3.2. Thread Object Contents

A thread needs several pieces of information in order to be scheduled on a processor. In addition to the parent object pointer, the address of the code representing the thread is required, as well as the value of the **`forall`** index or **`traverse`** pointer, if needed. A pointer to the enclosing Pascal procedure or function scope is also required. When a thread is scheduled for execution, the processor loads the Pascal scope pointer and thread object into registers, and branches to the code address held in the thread object. If the application code references the **`forall`** index or **`traverse`** pointer, the thread object pointer can be used to access its value. A typical thread object is shown in Figure 3.

If a thread creates child threads, two additional elements are required in the thread object. The first is a pointer to the *join point*: the address in the application code at which execution will resume after the children have synchronized. The second element is the *synchronization counter*, which is used to synchronize the child threads and track when the join is complete. Child threads access the counter via the parent thread pointer stored in their thread object. When synchronization is complete, the parent thread resumes execution at the join address held in the parent thread object.

4. Runtime Support Details

SpoC is a dynamically scheduled parallel execution environment. Concert does not provide hardware support for the parallel constructs in Simultaneous Pascal. In order to emulate these constructs, it is necessary to provide a package of runtime support functions which schedule work (represented by thread objects) on processors as an application executes. The efficiency of this software package is critical to the success of SpoC as a whole.

The principal measure of the effectiveness of the runtime system is the amount of overhead that is associated with a thread. This overhead includes all of the work which must be performed by the runtime system in order to create, schedule and synchronize threads. The amount of application work which can be performed by a thread must exceed the overhead associated with that thread in order to achieve effective scalability as processors are added to the system. Thus, the amount of overhead constrains the effective granularity of the parallel application. The smaller the overhead, the finer-grained the application can become.

4.1. Frames

The basic data object in the SpoC runtime system is the *frame*. A frame is a small piece of memory which is used to represent thread objects. A frame must contain all of the runtime system thread control variables, and must also provide space for the application defined local variables and the compiler determined induction variables and temporaries. The size of the frame needed by an application is determined by the compiler and is passed to the runtime system prior to execution. The runtime system then guarantees that all frames will be of sufficient size to execute the application.

A frame can be in one of three states: free, awaiting execution, or executing[†]. A free frame is kept in the free frame pool, and will be retrieved by a processor when it is needed to represent a thread. Once a frame has been used to represent a thread, it is placed into the work pool, to await execution by a processor. Finally, a frame is removed from the work pool, and holds the local thread state while the thread is executed by a processor. When the thread terminates, the frame is placed back into the free frame pool for subsequent reuse.

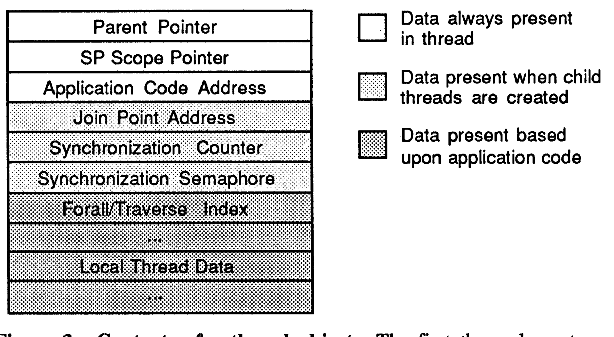


Figure 3. Contents of a thread object. The first three elements are required for a thread to execute. The next three are needed to track terminating child threads. The `forall` index or `traverse` pointer, along with any local variables, are required as dictated by the application code.

The rate at which frames can be placed into, and removed from, the free and work pools is a limiting factor on the speed of the runtime support software. The frame pools are shared data objects which must be accessed serially in order to preserve queue integrity, inducing logical contention in the runtime system. In addition, physical contention occurs as processors attempt to inspect the frame pools, which are stored in shared global memory.

The data structures that implement the frame pools must minimize both logical and physical contention. Contention is reduced by breaking the frame pool into several smaller subpools, which are accessed simultaneously by several processors in parallel. Physical contention is further reduced by exploiting the interleaved memory of the Concert hardware, and attempting to locate different elements of the pool in different memory banks.

The frame pools are implemented as a circular list of singly linked stacks of frames. Each stack has a semaphore to provide exclusive access, and a flag indicating if the stack is empty. There are as many stacks in the pool as there are processors in the system. Although all stacks are accessible by all processors, each processor is associated with a specific stack for purposes of frame insertion. The frame pool in a four processor system is shown in Figure 4.

A processor locates and removes a frame in the following manner. Beginning at its corresponding stack (processor P_i starts with stack S_i), the processor cycles around the circular list of stacks until a nonempty stack is found. The processor spin-waits on the stack semaphore until it acquires exclusive access to the stack. The processor then removes the top element of the stack, updates the empty flag (if needed), and resets the stack semaphore.

This scheme has several advantages. The use of multiple frame stacks helps to distribute the searching processors around the ring of stacks, reducing logical and physical contention. In addition, the use of the empty flag reduces the number of times a processor must obtain exclusive access to the stack in order to inspect it. Finally, the actual removal of the frame is quick, minimizing the amount of time the stack is exclusively owned by a processor. In reality, the entire frame removal process can be implemented with as few as fifteen MC68000 machine instructions; the critical section requires only five.

† Frames are also used to represent procedure and function scopes. Within this paper, the representation of routine scopes is not pertinent.

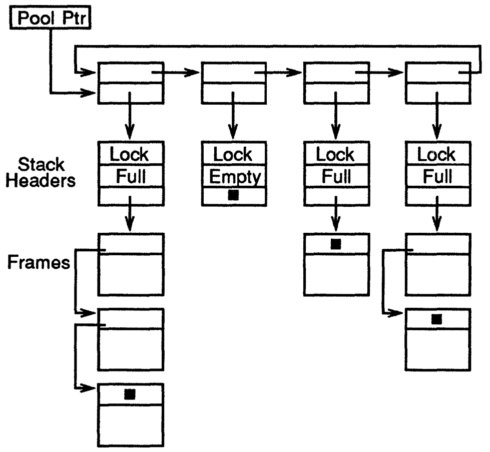


Figure 4. A four processor frame pool. Each processor retains a pointer to one of the four frame stacks. When a frame is required, a processor cycles around the stacks until a nonempty stack (indicated by the "Empty" flag) is found. The "Lock" field provides exclusive stack access. Frames are reinserted back into the stack indicated by the processor's pool pointer.

In order to insert a frame back into a pool, a processor P_i uses its own frame stack S_i. It spin-waits until exclusive access is acquired, places the frame on the stack, clears the empty flag, and releases the semaphore. This method reduces the time spent actually accessing the pool, since the processor does not have to search for a target stack for insertion. This routine can be implemented in as few as six instructions; the critical section requires only two.

4.2. Thread Creation

A thread goes through three phases during its life. It is created, it is executed, and it is synchronized and destroyed. Efficient implementation of each phase is important to achieve the fastest possible runtime system. In particular, communication between the compiler generated application code and the various runtime support routines must be as efficient as possible.

4.2.1. Fork Statement Expansion

The information required to expand a **fork** statement is the addresses of the statements in the body of the **fork**, and the join address, which is the address of the code to be executed following child synchronization. First, the join address is saved in the parent thread object. Then, the runtime routine iterates through the list of statement addresses, creating a child thread for each statement. As each child is created, a frame is obtained from the free frame pool, its fields are filled in, and it is placed in the available work pool. The last child thread is retained for execution by the processor that was expanding the **fork** statement. Typically, the number of children created by a **fork** statement is small (less than ten), and the overhead per thread for child creation can be as low as twelve instructions, plus the cost of frame acquisition.

4.2.2. Forall Statement Expansion

A **forall** statement requires less information for expansion than a **fork**, but can be more complex in implementation. The data required are the bounds of the range of the **forall** index, the address of the body of the **forall** statement, and the join address following the **forall** statement. Again, the runtime system first saves the join address in the parent thread object. It then creates a special **forall** control object, which holds the body address and the current and maximum **forall** index values. The processor expanding the **forall** immediately begins executing the body of the **forall**.

The compiler generates application specific code at the start of the **forall** body to determine if the **forall** has been completely expanded, based upon the current and maximum index values held in the **forall** control object. If expansion is not complete, the current index value is incremented, and the **forall** control object is placed back into the available work pool for another processor to retrieve and execute. The current processor then obtains a frame to represent the thread, initializes the appropriate fields in the frame, and continues with the body of the **forall** statement. If the current child thread is the last child of the **forall**, the processor simply keeps the **forall** control object and uses it as the frame for the child thread. Processor activity during **forall** expansion is shown in Figure 5.

This technique, although complicated, achieves a higher throughput in the creation of child threads. Unlike the more pedestrian technique of simply iterating across the **forall** index, creating all of the children at once, this technique allows several processors to be creating children simultaneously as the **forall** control object is passed around. If the children were all expanded at once, it is possible that the number of threads created would swamp the resources of the parallel machine. The incremental expansion of children uses only one additional frame to hold the **forall** control object while the child threads are executing, minimizing the use of machine resources. Finally, since the acquisition of a frame is left up to the processor executing the child thread, the processor can use local memory for the thread frame, reducing memory access time during thread execution. Although greater overhead is incurred by the incremental method, threads begin executing more rapidly, reducing the number of idle processors. Applications using the incremental approach finish faster than the equivalent application expanding **forall** statements using the iterative approach.

4.2.3. Traverse Statement Expansion

The **traverse** statement requires the programmer to specify a pointer to the head of a dynamic data structure, along with those fields in the structure which are pointers to other elements of the data structure. This information, along with the body and join addresses, is needed by the runtime system. Like **forall**, the **traverse** statement

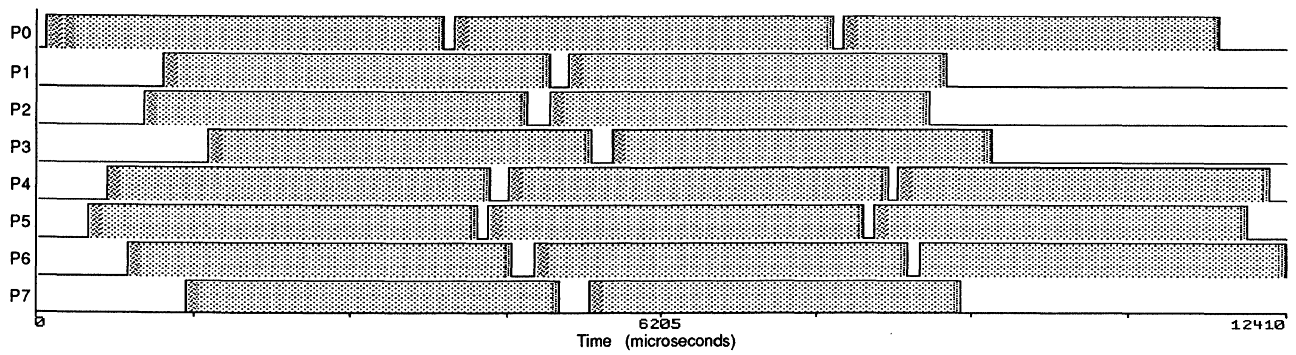


Figure 5. Processor activity during `forall` expansion. Here, eight processors expand and execute a twenty element `forall` statement. Each horizontal trace represents the activity of one processor. Each block within a trace represents the execution of a single thread; processors are otherwise idle. The darker region at the start of a thread is the `forall` expansion overhead; the darker region at the end is synchronization overhead. The remaining area is application work. Note that processors are performing useful work while expansion by other processors is occurring. Data was obtained using instrumentation hardware in Concert during actual application execution.

uses an incremental expansion technique. The only difference is that the compiler generated code at the beginning of the **`traverse`** body creates not one, but several threads, each thread corresponding to one of the programmer specified link fields in the data structure. In a **`traverse`** statement expanding across a binary tree, each processor would create child threads for the left and right children of the current element of the tree before processing the current tree node. These children would, in turn, create thread objects for *their* children before executing.

The **`traverse`** expansion achieves the same increase in throughput as the **`forall`** expansion, but does not avoid the excessive use of machine resources. Since each node can create a large number of child threads, many thread objects can be created as the data structure is traversed. In addition, the current implementation does not check to ensure that nodes in the data structure are visited only once, precluding the use of circular data structures. The overhead of **`traverse`** expansion depends upon the number of link fields specified by the programmer, but can approach the overhead of a **`forall`** expansion in a singly linked data structure.

4.3. Thread Execution

Once a thread has been created, it is placed in the available work pool. Idle processors in the system are constantly cycling through the work pool, trying to find a thread object to execute. When an object is found, the processor removes it from the pool and executes the code associated with the thread object.

A processor keeps pointers to the thread object and Simultaneous Pascal scope in registers during thread execution. References to the **`forall`** index or **`traverse`** pointer are made using the thread object address. Similarly, references to variables in the most local routine scope are made via the Simultaneous Pascal scope pointer. References to values in more global scopes are made by chasing the thread (or scope) parent pointers to the appropriate frame and retrieving the value. The compiler is aware of the cost of such pointer traversal, particularly into global memory, and attempts to cache global scope pointers in machine registers to improve access time. The various thread scope pointers are shown in Figure 6.

4.4. Thread Synchronization

When a thread finishes executing, it synchronizes with its sibling threads, and informs its parent of its termination. This is done by calling a runtime support routine which accomplishes the join operation.

This routine uses the parent pointer in the thread object to locate the parent thread object. As child threads were created, the synchronization counter in the parent thread was updated to reflect the number of executing children. As children terminate, they gain exclusive access to the parent thread object and decrement the child counter. When the counter reaches zero, no children remain, and the parent thread can resume execution at the join address.

Significant contention can occur when many children are attempting to access and modify the synchronization counter simultaneously. To reduce this conflict, counter access is made as fast as possible. Child threads can lock,

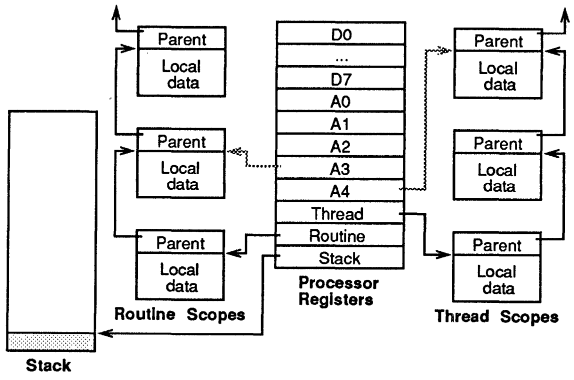


Figure 6. Scopes available to an executing thread. A thread retains dedicated pointers to its thread object and most local routine scope. Back pointers are used to find more global scopes; these pointers may be cached (gray arrows) as needed by the application. A thread also has access to a local stack for fast subroutine calls and interrupt handling.

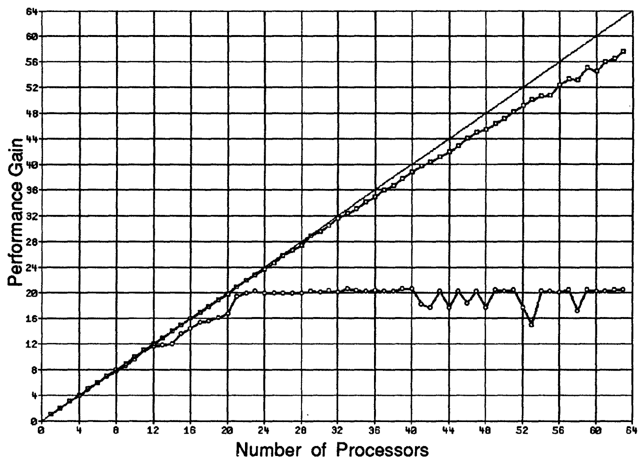


Figure 7. Scalability of the Mandelbrot application. The upper plot represents the pixel-wise version of the application; the lower plot, the row-wise version. The diagonal line is for reference; it represents linear scalability.

update, and release the parent thread object in as few as five instructions. Join contention can be further reduced by the nature of thread creation. Incremental thread creation tends to stagger the starts of sibling threads, which may stagger their termination, reducing simultaneous access to the synchronization counter.

When a processor detects that it has reduced the child count in the parent thread to zero, the parent can resume execution. The join address is fetched from the parent thread, various parent thread registers are restored from values saved in the parent thread object, and execution jumps to the join address.

5. Examples

The only way to evaluate the effectiveness of the SpoC system is to execute and analyze a variety of applications. The performance of any given application is dependent upon so many different factors that it is impossible to characterize the performance of SpoC using just one program. Although many applications have been executed on Concert via SpoC, the following two examples serve to demonstrate the range of performance obtained from the dynamically scheduled Simultaneous Pascal execution environment.

5.1. Mandelbrot

A Mandelbrot image[8] is a bitmap derived by iterating elements of the complex plane through the equation $z = z^2 + c$. The iteration completes when the magnitude of z exceeds two, or the number of iterations exceeds a predetermined limit. The number of iterations is then mapped into a set of colors, and the resulting image can be quite beautiful. Obviously, the desire for maximum resolution (providing more pixels per unit area in the image) and many colors (increasing the maximum iteration limit) results in a computationally intensive application.

Mandelbrot is naturally parallel at two levels. The algorithm can be expressed with row-wise parallelism, with each thread in the application computing all of the pixel values in a given row of the image. A more fine-grained approach exploits pixel-wise parallelism, creating a thread for each pixel in the image. Within SpoC, the former technique results in starvation, with too many processors idled by an insufficient number of threads. The latter technique results in an excellent example of a dynamically load-balanced parallel machine.

The scalability of the Mandelbrot application is shown in Figure 7. The upper curve shows the performance of the element-wise Mandelbrot calculation, while the lower curve represents the row-wise version. The tremendous locality of the application, coupled with the large work to overhead ratio in the threads, makes the element-wise version scale well across 64 processors. Information regarding thread sizes and overhead timing in the element-wise version of Mandelbrot is presented in Figure 8.

The row-wise version of Mandelbrot is no different from the element-wise in terms of locality, and has an even better work to overhead ratio. Unfortunately, the number of available threads is much smaller, and the resulting starvation causes scalability to top out at the twenty processor point. Once starvation sets in, the increased contention resulting from too many processors polling the empty work pool further contributes to loss of performance.

5.2. Gaussian Elimination

Gaussian elimination[9] is a classic technique for solving n equations in n unknowns. Gaussian elimination is a popular parallel application, since it is a naturally parallel algorithm. Like Mandelbrot, Gaussian elimination allows a wide range of granularity in the resulting parallel application. Unfortunately, in its SpoC implementation, Gaussian elimination suffers from a large number of references to a shared global data object. The contention resulting from these accesses makes Gaussian elimination a good example of scalability failure in the SpoC system.

Processor State	Entered	Time	Average	Percent
Awaiting Work	4225	137135	32.46	0.17
Thread Fetch	4225	150353	35.59	0.19
Executing	8581	79062232	9213.64	99.26
Expand Fork	0	0	0.00	
Expand Forall	4290	195051	45.47	0.24
Expand Traverse	0	0	0.00	
Perform Join	4290	108037	25.18	0.14
** Total **		79652808		

Figure 8. Breakdown of processor utilization in Mandelbrot. These figures, obtained from Concert instrumentation, show the time (in microseconds) spent in various states of the SpoC runtime system. The average times for scheduling primitives ("Expand Forall" and "Perform Join") show how quickly this task is accomplished in SpoC. The small "Awaiting Work" time represents processor idle time, and shows the excellent processor utilization in Mandelbrot. The average "Executing" time indicates the average thread length.

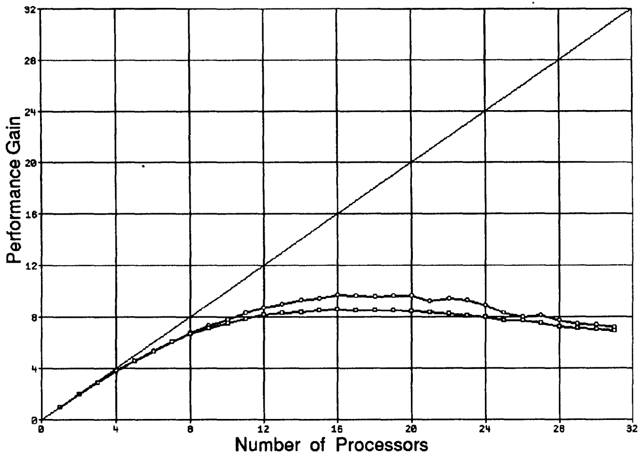


Figure 9. Scalability of Gaussian elimination. The upper plot represents the row-wise version of the application; the lower plot, the element-wise version. Excessive contention results in significantly degraded performance, regardless of granularity.

The graph in Figure 9 shows the scalability of two versions of Gaussian elimination. The upper plot is the row-wise version; the lower plot is the element-wise version. Regardless of granularity, both versions experience reduced performance improvement. The contention for access to the global data eventually overcomes any benefits obtained from adding additional processors. The contention and latency caused by global memory accesses in Concert increases thread execution times while interfering with ŞoC scheduling primitives. Even the best of optimizations within the ŞoC runtime system can be overwhelmed by an unfortunate conflict between a particular implementation of an application and artifacts in the Concert architecture.

6. Degradation

Loss of performance in a multiprocessor system can be traced to four principal sources. These sources are:

Starvation	is the time a processor is idle due to an insufficient number of executable threads.
Contention	is the time spent by a processor waiting for access to shared physical or logical objects.
Latency	is the additional delay of an access resulting from excessive distance in a distributed system.
Overhead	is the work performed managing parallel activities and resources not required in a sequential machine.

Starvation in a ŞoC application is a direct result of the programmer. As evidenced in the more coarsely grained version of Mandelbrot, a failure to create a sufficient number of appropriately sized threads in the parallel application will result in poor utilization of the underlying parallel hardware, reducing scalability. Although the ŞoC tool set includes tools to analyze the number and size of threads, the final responsibility for correcting this problem lies with the programmer, and not with the ŞoC runtime system.

Contention is a serious problem in the ŞoC system, at both the application and the system level. The use of shared data objects can result in excessive references to global memory. These references cause extraneous global bus traffic at the cluster and global memory levels, slowing unrelated accesses by other processors. ŞoC attempts to solve this problem in application programs by relying on smart compilation techniques to exploit locality. The contention problem within the ŞoC system software is minimized by judicious algorithm selection for the runtime routines, and much tuning based upon application execution data.

Latency in the ŞoC system is induced by artifacts in the Concert architecture, and there is little that can be accomplished in software to circumvent the additional cost of global memory references. As stated, every attempt is made to reduce the number of global memory references, minimizing the impact of memory access latency on ŞoC applications.

Overhead is the fundamental limiting factor in the performance of ŞoC applications. The time required to create, schedule, synchronize, and destroy threads presents a lower bounds on the granularity of ŞoC applications, and affects the ultimate scalability of the ŞoC system. Currently, each thread incurs approximately 140 instructions of overhead during its lifetime. Further improvements in software are unlikely, and hardware support is the only likely avenue of promise for future ŞoC performance improvements.

7. Implications for Future Architectures

Although a software implementation of the ŞoC runtime system has succeeded in producing a usable medium-grained parallel execution environment, further advances needed to support finer-grained applications will require support in the underlying machine architecture. These architectural features exploit specific aspects of the ŞoC system to reduce performance degradation.

The ŞoC experience has lead to the formulation of some basic ideas concerning the architecture of multiprocessors applied to general purpose parallel processing. The principal results obtained from ŞoC are:

- There is a limit to the performance gain that can be realized by implementing thread management primitives in software, indicating a need for thread support in hardware, and
- It is possible to support dynamically scheduled threads with a small number of well-defined prim-

itives whose distribution throughout the system critically impacts contention and latency.

This second result has significant implications in an advanced architecture. In SpoC, all primitives are performed by the microprocessor. In future architectures, support for specific primitives should be distributed throughout the system. For example, atomic frame management primitives should be part of a smart memory controller, not the microprocessor itself.

The driving premise of an advanced architecture for execution of a SpoC-like environment is to separate the general processor performing application work from the overhead of managing system parallelism. In the advanced SpoC architecture, this general processor would be augmented with a thread-context coprocessor and a structured memory controller. The processors would be linked to global memory via a prioritized, split transaction, general communication mechanism. A block diagram of this architecture is shown in Figure 10.

7.1. Thread-Context Coprocessor

The thread-context coprocessor is responsible for prefetching and initializing the context of threads. It acquires a thread object from the work pool stored in shared main memory and initializes the context of its associated microprocessor. Based upon compiler-supplied information, the coprocessor can prefetch code and static data for its companion microprocessor. When the thread completes, the thread-context coprocessor assists in updating the various control variables used to manage the SpoC parallelism tree.

From the perspective of the microprocessor, most of its accesses are to its local caches, initialized by its thread-context coprocessor. To the microprocessor, the apparent available bandwidth of the communications mechanism is greater since most of the transfers are done in advance by the coprocessor at low priority. Losses due to contention are greatly reduced, because fewer global accesses are made by the microprocessor. Those that do occur contend with fewer requests from other microprocessors.

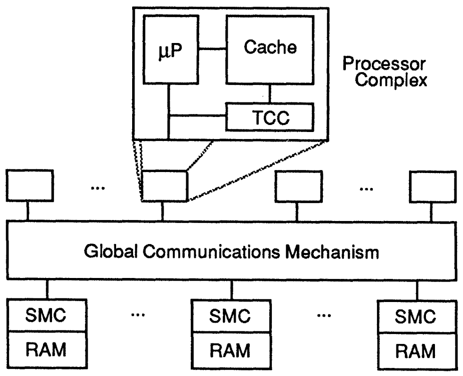


Figure 10. The Advanced SpoC Architecture. Processor complexes are connected to banks of global RAM via a global communications mechanism (GCM). The RAM supports primitive operations on high level objects via the structured memory controller (SMC). A processor complex consists of a microprocessor coupled to a high-speed cache and a thread-context coprocessor (TCC). The TCC prefetches code and data into the cache to increase the apparent processor/GCM communications bandwidth.

The introduction of a dedicated thread-context coprocessor significantly reduces the time required to create and initialize a thread prior to execution, as well as reducing the time required to synchronize and destroy a thread. The ability of the coprocessor to prefetch code and data into local caches minimizes the amount of global memory accesses, reducing contention and bus traffic in the memory communication mechanism.

7.2. Structured Memory Controller

The structured memory controller augments the conventional memory controller by providing the ability to manipulate the data structures employed by the SpoC runtime system. Such data structures include task queues, frames, and thread objects. Simple operations upon these objects are performed directly by the structured memory controller rather than the microprocessors. This reduces both contention on the communication channels and the time required to perform the functions.

For example, accessing the thread work pool could be defined as a primitive operation within the structured memory controller. A request from the thread-context coprocessor would cause the memory controller to locate and manipulate the appropriate pointers without further information from the processor complex. Analysis has shown that hardware support for thread queues would cut overhead in the current SpoC implementation by fifty percent, not including the positive effect derived from reduced global memory contention.

The structured memory controller also supports exclusive access to shared resources. Ordinarily, processors waiting for a locked logical resource poll the semaphore until it indicates the resource is available. This consumes communication and memory bandwidth, aggravating contention and degrading performance. The data structure representing a semaphore would contain as many bits as there are processors in the system. When a processor or coprocessor attempts to access a locked logical resource, its associated bit in the semaphore is set. When the resource is freed by the current user, the structured memory controller uses the set bits in the semaphore to select the next processor which will take possession of the shared resource. While the processor has lost time waiting for the object, increased contention due to polling is avoided.

8. Conclusions

General purpose parallel processing requires a parallel computing strategy for coordinating concurrent activities. Dynamic scheduling is essential for automatic load balancing for diverse applications. Medium to fine grain independently scheduled tasks are necessary for scalability and impose a need for low overhead mechanisms. The SpoC parallel execution environment has been presented and its implications for advanced parallel computer architecture have been discussed. SpoC executes programs written in Simultaneous Pascal, a version of Pascal augmented with explicit parallel constructs consistent with the structured nature of the original language. Simultaneous Pascal provides the programmer with the means for delineating the parallelism in the application algorithm. Portability of code to different multiprocessors is maintained by isolating the programmer from the actual machine organization.

The SpoC system supports general purpose parallel processing on the Concert Multiprocessor. Mechanisms for task creation, dispatching, and synchronization are realized in software. The type of parallelism used permits determination of join locations during thread creation for efficient synchronization. Locality of reference within threads is exploited using local memory for each processor to reduce global memory access and contention for shared communication resources. Tasks on the order of 140 instructions can be effectively handled by SpoC, providing substantial parallelism for many applications.

The SpoC system has provided insight into the features of an advanced parallel architecture suitable for executing Simultaneous Pascal applications but exhibiting superior scalability. This architecture reduces overhead, allows finer-grained tasks, and minimizes losses due to contention. This is accomplished by separating application execution from thread management, providing maximum utilization of the processors for application work. A dedicated coprocessor prefetches and manages thread objects, removing this responsibility from the application processor. Smart memory controllers in global memory perform compound operations on structured data elements. Memory controller support for runtime functions greatly reduces their execution time and reduces communication traffic along with its resulting contention.

The SpoC system has shown that a scalable parallel processing environment can be realized in software. By defining a specific parallel virtual machine, and implementing that virtual machine on a general purpose multiprocessor, SpoC has provided general purpose medium-grained parallel processing to its users. The insight gained from executing a variety of applications on SpoC has allowed its developers to propose a set of components which, if incorporated in future multiprocessors, would provide significant performance gains for the SpoC parallel virtual machine.

Acknowledgments

The authors wish to recognize the significant contributions made by Ellery Y. Chan, Douglas A. Thomae, Michael D. Noakes, and Kenneth C. Laprade to the realization of the SpoC parallel execution environment. In addition, the many helpful comments by other members of the Harris Advanced Technology Department, and the Parallel Processing Group at the Laboratory for Computer Science at MIT have proved invaluable in making SpoC a reality.

References

[1] T.L. Sterling, A.J. Musciano, E.Y. Chan, and D.A. Thomae, "SpoC: An Effective Implementation of a Parallel Language on a Multiprocessor," *IEEE Micro*, Vol. 7, Num. 6, Dec. 1987, pp. 46-62.

[2] T.L. Sterling, "Parallel Control Flow Mechanisms for Dynamic Scheduling of Tightly Coupled Multiprocessors", *Ph.D. Thesis*, EECS Dept., MIT, May 1984, pp. 24-29.

[3] R.H. Halstead, T.I. Anderson, R.B. Osborne, and T.L. Sterling, "Concert: Design of a Multiprocessor Development System," *13th Symposium on Computer Architecture*, June, 1986, pp. 40-48.

[4] P.C. Treleaven and R.P. Hopkins, "Decentralized Computation," *Eighth Symposium on Computer Architecture*, May, 1981, pp. 279-290.

[5] D. Cooper, *Standard Pascal User Reference Manual*, W.W. Norton & Company, 1983.

[6] M.J. Flynn, "Very High-Speed Computing Systems," *Proc. IEEE*, Vol. 54, 1966, pp. 1901 - 1909.

[7] J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, Vol. 21, No. 8, August 1978, pp. 613 - 641.

[8] H.-O. Peitgen and P.H. Richter, *The Beauty of Fractals*, Springer-Verlag, 1986, pp. 151-160.

[9] R. Beckett and J. Hurt, *Numerical Calculations and Algorithms*, McGraw-Hill Book Co., 1967, pp 88-90.

A SHARED MEMORY ALGORITHM AND PERFORMANCE EVALUATION OF THE GENERALIZED ALTERNATIVE CONSTRUCT IN CSP[1]

Hwa-chung Feng
AT & T
200 Laurel Avenue
Middletown, NJ 07748

Richard M. Fujimoto
Department Of Computer Science
University Of Utah
Salt Lake City, UT 84112

Abstract

Communicating Sequential Processes (CSP) is a paradigm for communication and synchronization among distributed processes. The alternative construct is a key feature of CSP that allows nondeterministic selection of one among several possible communicants. Previously, an algorithm for *shared-memory* multiprocessors was developed for the generalized version of Hoare's alternative construct that allows output commands to be included in guards. Here, the performance of this algorithm, as implemented on the BBN Butterfly Parallel Processor, is evaluated. An adaptive approach that automatically optimizes the performance of the synchronization mechanism is suggested and evaluated.

1 Introduction

Communicating Sequential Processes is a well known paradigm for parallel computation [5,6]. A CSP program consists of a collection of processes that interact by exchanging *messages.* The message-passing primitives, called input and output commands, are synchronous. An important feature of CSP is the *alternative* construct which is based on Dijkstra's guarded command [2]. This construct enables a process to *nondeterministically* select one communicant among many.

In [4] an algorithm was proposed for implementing the *generalized* alternative construct that allows output commands to be included in guards. A variation of this algorithm was implemented on the BBN Butterfly^{TM} multiprocessor. The principal contribution of this paper is to present an empirical performance evaluation of this algorithm and to suggest techniques to optimize its performance.

For completeness, we will first describe the algorithm and its implementation. A correctness proof and more detailed discussion of the algorithm appears in [4]. The performance evaluation studies follow.

2 The Alternative Algorithm

A CSP program contains processes P_1, P_2, ..., P_N. Process P_i is assigned the unique *process ID i* to distinguish it from others.

Each invocation of an alternative operation is referred to as a *transaction* that begins when the operation is initiated and ends when a successful communication has been completed. A process will usually engage in many transactions during its lifetime. A total ordering is imposed among all transactions entered by *all* processes of a given CSP program. A unique sequence number, referred to here as a *transaction ID,* is associated with each transaction.

Two processes that initiate alternative operations that result in a communication between them are said to *rendezvous.* In a *typical* rendezvous, the first process to enter the alternative will block and wait for a signal from the second. When the second process enters the alternative, it will *commit* to the first in order to obtain "permission" to rendezvous; the "committing" process will then signal and exchange a message with the blocked process, and both will complete their respective alternative operations. The algorithm uses an "abort and retry" mechanism to avoid race conditions when two potential communicants simultaneously enter the alternative command.

[1]This work was supported by ONR contract number N00014-87-K-0184 and NSF grant number DCR-8504826.

2.1 Primitive Operations

The machine is assumed to be a shared memory multiprocessor. To simplify the presentation, we will assume certain synchronization primitives are available. Each can be easily constructed using a test-and-set and standard scheduling primitives. In particular, we will assume the following are available:

- **AtomicAdd(X): INTEGER** atomically increments the integer variable X and returns the original value of X.
- **Lock** and **Unlock** provide exclusive access to shared data structures.
- **WaitForSignal** and **Signal** block and unblock the process, respectively. A signal contains a single, user-defined integer value. If a signal is not absorbed before a second one arrives, it is discarded.
- **Send(M, R)** and **Recv(R): Message** provide the blocking send and receive function, where **R** is the remote process ID.
- **Sleep(T)** causes the process invoking it to block for at least T time units. A process will always eventually awake after calling *Sleep.*

2.2 Process States and Shared Variables

Each process can be in one of the following states:

- **WAITING.** The process is blocked on a *WaitForSignal* operation, waiting for a rendezvous.
- **ALT.** The process has begun an alternative operation and is scanning through its guard list to find a possible rendezvous.
- **SLEEPING.** The process was forced to abort an alternative operation. After aborting, the process goes to sleep for some time before retrying.
- **RUNNING.** The process is executing code not related to the alternative operation.

A state transition diagram for each process is shown in figure 1. The ALT and SLEEPING states should be viewed as "transitory" states through which a process must pass while trying to commit or move into the WAITING state. A process cannot remain in either the ALT or the SLEEPING state for an unbounded amount of time on a single transaction [4].

Each process P_j maintains a number of variables that may be examined, and in some cases modified, by other processes:

- **AltList$_j$** lists the guards associated with the last alternative operation initiated by P_j that caused P_j to enter the WAITING state.
- **AltLock$_j$** is a lock used to control access to *AltList$_j$*. It is initialized to 0 (unlocked).
- **State$_j$** holds the current state of P_j. It may be set to WAITING, ALT, SLEEPING, or RUNNING, and is initialized to RUNNING.
- **WakeUp$_j$** is initialized to 1 and is set to zero by P_j whenever it enters the WAITING state. It is incremented (atomically) by processes trying to commit to P_j.

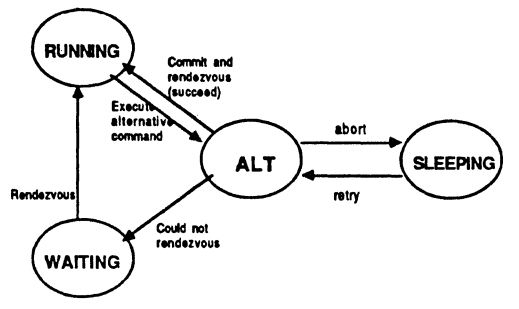


Figure 1: The state diagram of a process.

A system wide global variable, **NextTransID**, is initialized to zero and is atomically incremented each time a process initiates an alternative operation. If necessary, this can be implemented using local variables and unique processor IDs to avoid contention [4].

A procedure **CheckAndCommit(AltList$_r$, g$_i$): INTEGER** is called by process P_l (l denotes the *local* process) to check that "valid" communications can take place between P_l using guard g_i and P_r (r denotes the *remote* process), and if so, to attempt to commit to P_r. If a commit was attempted and succeeded, *CheckAndCommit* returns a value indicating the corresponding guard in the *remote* process P_r. Otherwise, FAILED is returned.

2.3 Other Notation

For notational convenience, other variables and predefined functions are defined that are used in the algorithm. These include:

- **TransID$_l$** is a variable that contains the ID of the current transaction in which process P_l is engaged.
- **CommunicantID(g$_i$)** is a function that returns the ID of the process listed in the I/O command portion of guard g_i.
- **Communicate(g$_i$)** executes the I/O command in guard g_i.
- **TimeOut** is a constant indicating the period of time a process should sleep after an aborted attempt.

The algorithm is shown in figures 2 and 3. The *Alternative* procedure shown in figure 2 is a "front end" that is responsible for retrying aborted attempts. The heart of the algorithm lies in the *TryAlternative* procedure shown in figure 3. The parameters passed to both procedures are n enabled I/O guards $g_1, g_2, \ldots, g_n$. Each guard contains either a single output or a single input primitive. This procedure does not return until a rendezvous has been completed at which time it returns an integer indicating the guard ($g_1, g_2, \ldots, g_n$) that was eventually satisfied. See [4] for a more complete description of the algorithm.

3 Performance Evaluation

Because the algorithm relies on an abort-and-retry mechanism to avoid race conditions, a performance analysis is required to ensure it does not thrash under certain "stressed" workload conditions. An implementation of the algorithm was developed for a 16-node BBN Butterfly Parallel Processor for this purpose. The implementation was written in C.

3.1 Butterfly Hardware

The BBN Butterfly multiprocessor contains up to 256 processing nodes and a high performance interconnection switch. Each processor node contains a 16 MHz MC68020 with MC68881 floating point coprocessor, up to 4 MBytes of memory, and a *processor node controller* (PNC), a microcoded engine implemented with 2900 series AMD parts. The interconnection switch is configured as an Omega network [1].

All memory references made by the 68020 are passed to the PNC. Local memory references are forwarded to the local memory, while remote references are passed to the appropriate processor node through the switch. A local reference requires approximately 600 nanoseconds and a remote reference 4 microseconds assuming no switch contention. The PNC also handles memory requests made by other processors to this node, as well as atomic memory operations.

```
/* g_i are enabled I/O guards */
PROCEDURE Alternative(g_1, ..., g_n):  INTEGER;
VAR
    INTEGER ReturnValue; /* indicates guard which rendezvoused */
BEGIN
    /* l is the local process id */
    TransID_l := AtomicAdd(NextTransID);
    ReturnValue := FAILED;
    WHILE (ReturnValue = FAILED) DO
        ReturnValue := TryAlternative (g_1, ..., g_n);
        IF (ReturnValue = FAILED) THEN Sleep (TimeOut); END;
        END;
    RETURN (ReturnValue);
END Alternative;
```

Figure 2: The "front end" procedure. *TryAlternative* returns the number of the guard on which a rendezvous took place or FAILED if it aborted.

3.2 Factors and Metrics

The time that a process spends in a specific alternative operation is affected by many factors. The number of guards affects the amount of time required to scan the guard list. The amount of computation that a process conducts between two consecutive alternative operations also influences performance. The more frequently processes enter the alternative operation, the more likely collisions are to take place, potentially increasing the number of aborted operations, and reducing performance.

The sleeping period, i.e., the amount of time a process waits after an alternative operation aborts, is an important parameter that must be set to an appropriate value. A sleep period that is too short may cause a process to wake up when its neighbors are still in the ALT state, leading to additional aborted attempts and thrashing. On the other hand, an excessively long sleeping period could lead to an unnecessary delay while the process remains in the SLEEPING state. An adaptive

```
PROCEDURE TryAlternative(g_1, ..., g_n):  INTEGER;
VAR
    BOOLEAN flag;
    INTEGER GuardNumber; /* corresponding guard of P_r */
    INTEGER i, r;
BEGIN
    State_l := ALT;
    /* look for rendezvous with a waiting process.  */
    FOR i:=1 TO n DO
        r := CommunicantID(g_i);
        flag := TRUE;
        WHILE (flag) DO
            CASE State_r DO /* The remote process state.  */
            SLEEPING: flag := FALSE;
            RUNNING: flag := FALSE; /* try next guard */
            WAITING: GuardNumber := CheckAndCommit(r, g_i);
                IF (GuardNumber = FAILED) THEN
                    flag := FALSE; /* try next guard */
                ELSE /* Wake up P_r */
                    State_l := RUNNING;
                    Signal(r, GuardNumber);
                    Communicate(g_i);
                    RETURN (i);
                END;
            ALT: IF (TransID_l > TransID_r) THEN
                    State_l := SLEEPING;
                    RETURN (FAILED); /* abort...*/
                ELSE /* busy wait loop.  */
                    WHILE ((State_r = ALT) DO END
                END; /* if-then-else */
            END; /* case statement */
        END; /* while loop */
    END; /* for statement */
    /* couldn't find guard to rendezvous */
    Lock(AltList_l); AltList_l := (g_1, ..., g_n); Unlock(AltList_l);
    WakeUp_l := 0; /* first to commit gets rendezvous */
    State_l := WAITING;
    i := WaitForSignal(); /* Blocks */
    State_l := RUNNING;
    Communicate(g_i)
    RETURN (i);
END TryAlternative;
```

Figure 3: The *TryAlternative* procedure attempts to rendezvous with a process listed in an I/O guard, and does not return until rendezvous takes place.

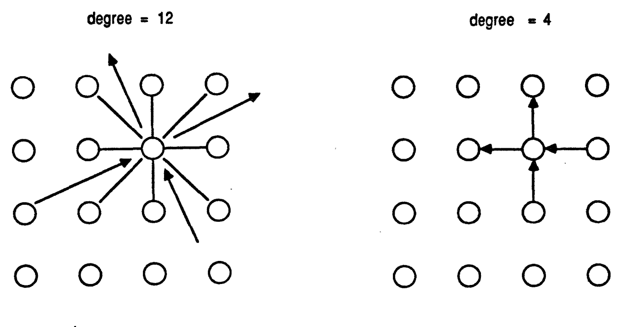


Figure 4: Typical meshes (degrees 4 and 12).

technique will be described later to assign an appropriate value for the sleeping period.

3.3 Test Programs

A synthetic workload program was designed to determine appropriate methods for setting the sleep period. The parameters for this program are:

topology The channel connection pattern among the processes. This determines the size of the guard list in the alternative operation.

interval computation The amount of time each process spends in the RUNNING state between consecutive alternative operations.

sleep period The time a process spends in the SLEEPING state when it aborts.

The size of the message that is exchanged is not considered here because the communication takes place *after* the rendezvous point is reached, so it is in fact not part of the alternative operation and may be viewed as part of the interval computation.

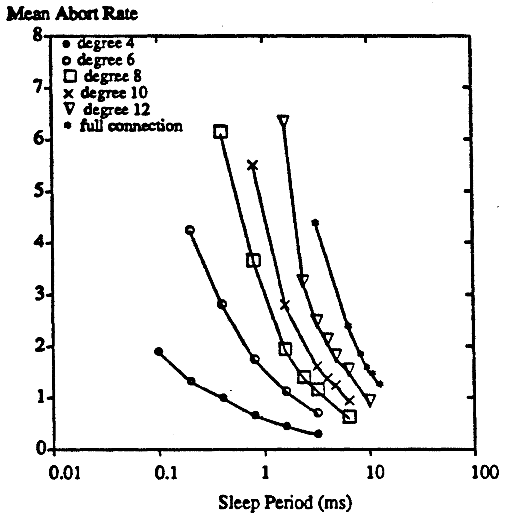


Figure 5: Abortion rate with no interval computation.

The test programs used for these experiments were configured as a lattice of 16 processes, each communicating with some number of its neighbors. This number is referred to as the *degree*. Figure 4 shows the connection patterns of meshes with degrees 4 and 12. The programs used in these experiments are meshes with degree 4, 6, 8, 10, 12, and a full connection pattern, i.e., degree 15. The *symmetric* mesh connection pattern is chosen to avoid *bottlenecks* that might bias the result. Wrap-around connections are used at the edges at the mesh to maintain symmetry.

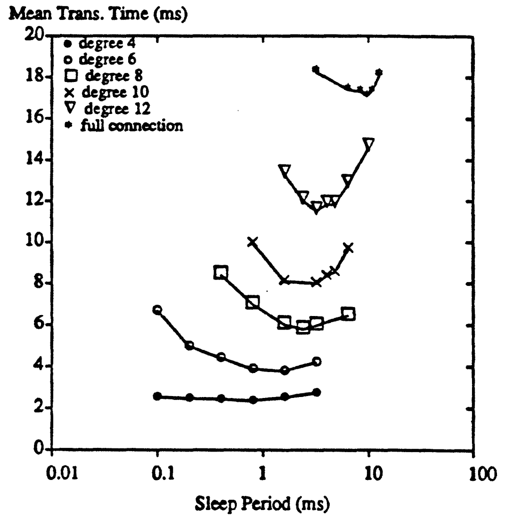


Figure 6: Transaction time with no interval computation.

Each of the 16 processes repetitively executes an alternative operation, attempting to rendezvous with one of its neighbors. Each process is mapped to a separate processor to maximize the likelihood of collisions. These experiments study the *worst case* behavior of the alternative operation under different parameter settings. Performance in a *typical* application will be discussed later.

The length of interval computation is chosen from an exponential distribution with a mean ranging from 500 microseconds to 16 milliseconds. The sleep period ranges from 100 microseconds to 12.8 milliseconds. The performance metrics for these experiments are (1) the average transaction time, and (2) the abortion rate, i.e., the average number of abortions per transaction. The reported measurements are mean values with a 95 percent confidence interval about the calculated mean value of less than 4 percent.

All of the time measurements are derived from recordings of the real time clock which has a resolution of 62.5 microseconds, and include operating system overhead. It is estimated that the total error is within 5 percent of the reported results. To minimize interference, the measurements were taken with no other user processes on the machine.

3.4 Results

Figure 5 shows the abortion rate when the interval computation is zero, maximizing the probability of abortion. In topologies with low degree, e.g., 4 and 6, the abortion rate is not affected as dramatically as it is for large degree topologies as the sleeping period is varied. This is because the probability of seeing each other in the ALT state is already small for topologies with small degree.

Figure 6 shows the mean transaction time with zero interval computation. Each topology has a different "best" sleeping period that increases with the degree of the topology. When the list of guards becomes longer, the alternative algorithm tends to spend more time scanning the list, and therefore spends more time in the ALT state. As a result, a longer sleeping period is needed to avoid excessive abortions. However, an excessively long period is undesirable as well.

Table 1 shows the approximate "best" sleeping periods in each topology in milliseconds, along with the average abortion rate. The percentage in parentheses indicates the standard deviations of the respective mean values. They suggest that topologies with large degree have large variances.

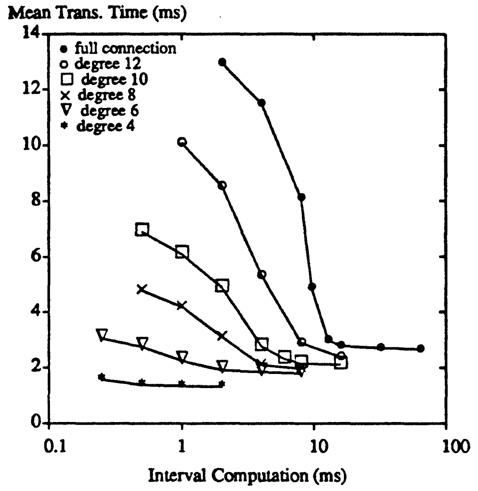


Figure 7: Transaction time with non-zero interval computation.

Figure 7 shows the average transaction time as a function of the mean interval computation time when the sleep period is set to the "best" value for each topology. The interval computation spreads alternative operations attempts over a longer period of time, thereby reducing the frequency of conflicts. Because it achieves the same effect as a longer sleep period, a similar relationship between the interval computation and the performance of the algorithm is expected. The more sparsely an application executes the alternative operation, the closer the performance of each operation is to the optimum. Not surprisingly, the least amount of interval computation necessary to achieve optimal performance, i.e., zero abortion, is comparable to the "best" sleep period.

3.5 An Adaptive Approach to Setting the Sleeping Period

The results of the previous section indicate that an optimal sleeping period for one configuration is far from optimal for others. Therefore, no *single* sleep period is appropriate for all situations. To accommodate a wide variety of circumstances, an *adaptive* approach of setting the sleeping period is required.

One measure for determining when the sleeping period should be reset is the number of abortions the current transaction has committed so far. Each additional abortion in a given transaction indicates that the current sleeping period may be too short. The following scheme is proposed to dynamically set the sleeping period: The sleeping period is initially set to a small value on each transaction. During the lifetime of this transaction, each additional abortion causes the sleep period to be *doubled*. This strategy is not unlike the algorithm used in the Ethernet local area communication network to resolve collisions [7].

Although large degree topologies may require several abortions before a suitable sleep period is accumulated, the exponential approximation should quickly converge to an appropriate value. Table 2 validates this intuition. The performance of the adaptive scheme is seen to match that obtained when the sleep period was manually optimized for each particular configuration. The sleeping period in these experiments was initially set to 400 microseconds at the beginning of each transaction. In topologies with large degrees, the improvement is especially apparent because many transactions now spend much less time sleeping.

In addition to the experiments based on synthetic workloads, the performance of a "typical" application program was also investigated. The *bounded buffer* program [3] consists of a buffer process and some number of producer and consumer processes. The producer processes repetitively generate data and send them to the buffer process, while the consumer processes consume the data collected by the buffer process. The buffer process can hold up to four unconsumed data items before it disables all the channels connected with the producers. Similarly, the channels with the consumers are disabled if there are no data to be consumed. When executed on the Butterfly, it was observed that the buffer process only spends an average of 600 microseconds to reach rendezvous regardless of the number of producers and consumers. There are very few aborted attempts. This result is much better than the previous "worst case" performance and can be explained as follows. All of the producers and consumers spend most of their time in the WAITING state because the buffer can rendezvous only one process on each repetition. Therefore, the buffer process will usually rendezvous with the first process it scans every time it enters an alternative operation, minimizing the transaction time.

Table 1: "Best" Sleeping Period with no Interval Computation.

Degree	Avg. Abort Rate	Avg. Tran. Time	Sleep Period
4	(10%) 0.9	(25%) 2.0	0.6
6	(8%) 1.0	(58%) 3.3	1.2
8	(6%) 1.6	(67%) 5.1	2.4
10	(6%) 1.7	(75%) 7.2	3.2
12	(4%) 1.8	(78%) 10.4	4.8
fc	(5%) 1.5	(92%) 17.2	9.6

Table 2: Performance of the Adaptive Scheme.

degree	abortion/tran	time(ms)/trans
4	0.3	1.8
6	1.6	3.3
8	1.9	4.8
10	2.5	6.9
12	2.9	9.7
fc	3.5	12.9

4 Conclusions

We have presented an algorithm that implements the generalized alternative construct in CSP. Unlike previous algorithms, it is based on a shared memory architecture. An implementation, written in C, has been developed for a 16-processor BBN Butterfly and extensive performance evaluations were conducted. An important parameter of the algorithm is the amount of time that the process waits after each aborted attempt before attempting to retry the operation. It was found that the appropriate sleep period is application dependent, so an adaptive scheme was suggested to dynamically set this parameter. Empirical data indicate that this approach is effective in practical situations.

References

[1] B. Thomas, et al. *Butterfly Parallel Processor Overview.* BBN Report No. 6148, BBN Laboratories Incorporated, March 1986.

[2] E. W. Dijkstra. Guarded Command, Nondeterminism and Formal Derivation of Programs. *CACM*, 18(8):453–457, August 1975.

[3] R. E. Filman and D. P. Friedman. *Communicating Sequential Processes*, chapter 10. Computer Science Series, 1984.

[4] R. Fujimoto and H.C. Feng. A Shared Memory Algorithm and Proof for the Alternative Construct in CSP. *International Journal of Parallel Programming*, June 1987.

[5] C. A. R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, August 1978.

[6] C. A. R. Hoare. *Communicating Sequential Processes. Computer Science*, Prentice Hall, 1985.

[7] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *CACM*, 19(7):395–404, July 1976.

DESIGN STRATEGIES FOR THE RUN-TIME SUPPORT TO THE ADA RENDEZVOUS

Silvano Rivoira
Dipartimento di Matematica - Università di Perugia
Via Vanvitelli, 1
06100 Perugia (Italy)

Abstract -- This paper describes an experimental comparison of three different strategies for the design of the run-time support to the tasking facilities in the ADA language. The three strategies differ for the task in whose thread of control the accept body is executed during a rendezvous: the called task (server model), the calling task (procedural call model) and the last task joining the rendezvous (order of arrival model). All models have been implemented in the operating system kernel of a tightly-coupled multiprocessor system based on 16 bit microprocessors. The time requirements of the alternative implementations have been measured and the results are discussed in the last section. Finally the efficiency of the rendezvous models has been compared with that of the semaphoric model in the solution of the classical bounded buffer problem.

Introduction

The programming language ADA [1] contains powerful tasking facilities for concurrent and real-time programming, which allow a programmer to concentrate on parallel system design and to ignore inter-task synchronization and communication details.

The strength of the rendezvous (RV) mechanism is that it unifies the semantics of different multiprocessing construct, providing a very general, expressive, elegant means for developing parallel systems. However, the implementation of these high level facilities, due to their generality , raises significant problems to the designers of compilers and run-time supports for ADA [2] - [3]. One of the main problems is how to avoid excessive scheduling interactions in many synchronization cases which frequently occur in real-time applications [4].

Considerable research and experimentation efforts have been devoted to propose implementation strategies of RV [5] - [6] - [7] - [8] - [9] - [10], to compare the ADA's tasking facilities with those of other languages [11] - [12], and to explore the performance of different implementations of RV on a given machine under constrained circumstances [13].

Unfortunately it is often difficult, or even impossible, to compare solutions and experiences from different authors, because they have been based on different assumptions or obtained for different architectures.

The goal of this work is to evaluate the efficiency of different implementations of the ADA parallel facilities on monoprocessor and multiprocessor architectures, and to compare it with that of other communication and synchronization mechanisms. A modular architecture (MODIAC) developed by the Consiglio Nazionale delle Ricerche of Italy and proposed as a national standard for real-time control of continuous and discontinuous industrial processes [14], is used as a benchmark. The architecture is organized as a tightly-coupled multiprocessor system based on 16 bit microprocessors, and it currently supports a distributed.Operating System kernel (MODOSK) which extends the programming language PASCAL with synchronization, message passing and short time scheduling primitives between processes [15].

Three different implementation strategies of the ADA tasking facilities for such an architecture are considered. The implementations have been derived from the ADA rationale [5] and from a formal model of the ADA multitasking constructs developed as part of a total formal description of the language [16].

The ADA kernel is designed as a set of primitives identically replicated in the private memory of each processor. The primitives control the concurrent execution of the tasks running on the processor pool and they allow interactions between tasks independently of their physical allocation.

The different RV implementations have been coded in the sequential PASCAL language, excepting only a few functions of the lowest level, coded in assembly language. The ADA tasking facilities are supported by a set of kernel primitives which can be invoked as procedures by tasks also implemented as PASCAL procedures.

The use of a high level system language allowed us to reduce the development time and to produce more readable and portable code. Using PASCAL has also the aim, in this case, of comparing the efficiency of the ADA concurrency constructs with respect to the efficiency of the synchronization mechanisms provided by MODOSK in the same environment. For this reason the MODOSK and ADA kernel primitives use similar data structures and similar implementations for ready queues, time schedule queues and task descriptor management.

The following sections describe: the main features of the alternative implementations of the RV mechanism; the kernel structure; the translation scheme of the ADA constructs; the operations involved in the order of arrival implementation. Finally the results of the time requirement measurements are reported in the last sections.

Alternative Implementations of the RV Mechanism

In the RV implementation suggested by the ADA rationale [5] and referred to in this paper as "server" RV, the calling task remains suspended until the called task executes the accept body. It is worth noting that, in order to complete a RV, the scheduler is invoked (and possibly a context switch occurs) two times in the case entry calling precedes the execution of the accept statement and three times in the other case. One or two interprocessor interrupt signals and two or four scheduling operations respectively are necessary if the interacting tasks are running on different processors.

This situation is shown in Fig. 1a, where scheduling points and interprocessor interrupts are represented by dots and double arrows respectively. A single copy of the accept body is sufficient and it can be stored in the private memory of the processor running the accepting task. Parameter passing may be performed through the shared memory.

A second approach, the "procedural call" rendezvous [7] - [8], states that the accept body is always executed by the calling tasks. Task interactions for this case are shown in Fig. 1b. The advantage of this implementation is that no special mechanism is necessary for parameter passing because the caller executes the accept body in its thread of control. The disadvantage is that the accept body must be accessible to the calling tasks. Accessibility can be obtained either by replicating the code of accept bodies in the private memory of each processor running a calling task, or by storing the code in the shared memory.

Since the accept body can refer to global variables declared in a larger environment, these variables must be referable by different tasks, possibly running on different processors, and therefore they also have to be stored in shared memory. This solution becomes ineffective or impossible when some resource requested by an accept body is only available on a particular processor.

A solution which reduces the number of scheduling points during a RV consists of

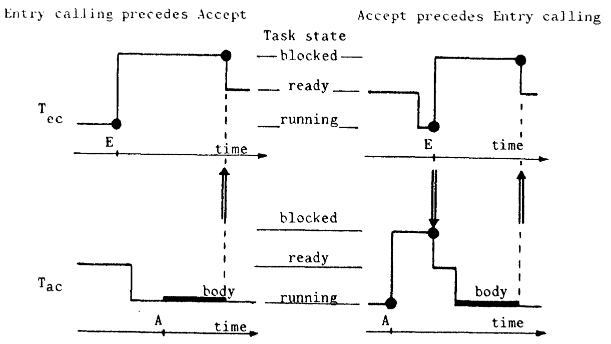


Fig. 1a - Server rendez vous

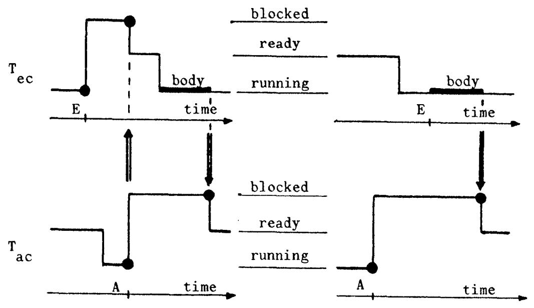


Fig. 1b - Procedural call rendez vous

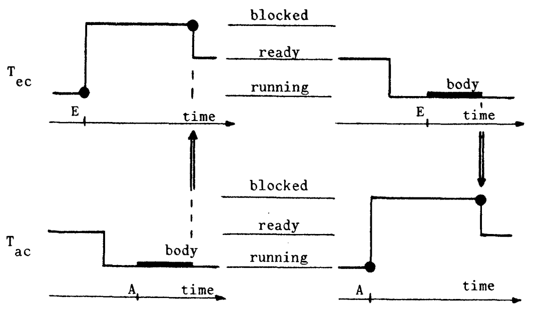


Fig. 1c - Order of arrival rendez vous

Fig. 1 - State transitions of the calling (T_{ec}) and accepting (T_{ac}) tasks during a rendezvous

executing the accept body as a part of the thread of control of the last task which joins the RV. This approach can be referred to as the "order of arrival" implementation and it has been proposed by several authors [6] - [7] - [9]. It appears to be effective both on monoprocessor architectures (where it reduces the scheduling points to two) and on tightly coupled multiprocessor architectures, where only one interprocessor interrupt signal and two scheduling operations are needed to complete a RV (Fig. 1c). This solution, however, shares with the procedural call implementation the difficulties due to global variable references and private resource access.

At this time it is unknown whether the performance gain due to fewer context switches compensates for the overhead introduced by such difficulties.

One of the aims of this work is to contribute in answering this question.

Structure of the ADA Kernel

As previously mentioned, the experimental analysis has been performed on a tightly coupled multiprocessor architecture based on 16 bit microprocessors (Zilog, Z8001).

The system has a multiple bus interconnection scheme with three bus levels: private, shared and global [14]. Physical resources connected to a private bus are accessible only by the processor which owns the bus; resources connected to the global bus are accessible by every processor in the system; resources connected to a shared bus can be directly accessed (that is without competition) by one processor and indirectly accessed (through the global bus) by all the other processors. The arbitration mechanism of the global bus is distributed and the contention is resolved on a single transaction basis. The indivisible read-modify instructions required to allow lock operations on common variables are obtained by inhibiting the access to the global bus for one or more transactions. Each processor can send interrupt signals to any other processor; more requests can be sent at the same time to a same processor.

In the following we shall assume that the instruction code executable by a processor resides in the processor private memory and that some data structures are located in shared memory, where they can be referred to by tasks allocated on different processors. Tasks descriptors are divided into two parts: local and global.

The local descriptor (LTD) resides in the private memory of a processor and it contains all the information necessary to the execution of the task concurrently with other tasks resident in the same processor. The global descriptor (GTD) resides in the shared memory directly connected to the processor which owns the task, and it contains the information which allow the interactions between tasks allocated on different processors. The kernel is constituted by a set of primitives identically replicated in the private memory of each processor.

When a task invokes the kernel to interact with another task, the invoked primitive (executed by the processor which owns the calling task) checks for the processor identifier in the global descriptor of the called task. If the called task resides in the same processor as the calling one, all the information about its status is available in the local descriptor and the requested operation can be performed. If the called task resides in a different processor, the kernel primitive sends an interrupt request to that processor, together with the code of the requested operation, and the references to the called task and the calling processor. The interrupt service procedure resident on the interrupted processor will then invoke the kernel primitive corresponding to the requested operation in order to complete the task interaction.

It is worth noticing that a clever allocation of a logical resource at the appropriate bus level can greatly improve the system's performance. This is particularly true for the RV implementations (procedure-call and arrival-ordered) requiring that some variables declared in one environment are accessible by tasks possibly running on different processors.

The main content of a local task descriptor (LTD) is the following:

- a statework S referring to the hardware and software context of the task;
- a status field, which may assume the following values:
  - RUNNING: the virtual processor (VP) has the control of the processor;
  - READY: the VP is eligible for processor utilization on the basis of its priority;
  - BLOCKED: the VP does not compete for processor utilization because it is scheduled for a later time or because it is waiting for an event which will be caused by a partner task;
- the task priority;
- the scheduling time, that is the time value set by a delay statement;
- a link to another LTD;
- a link to the GTD of the same task.

A global task descriptor (GTD) essentially contains the following information:

- the processor number;
- a lock variable;
- a status field, which assumes the ACTIVE value when the corresponding LTD status is RUNNING or READY, while it assumes one of the following values when the LTD status is BLOCKED:

ENTRY CALLING:	the task is blocked as a consequence of an <u>entry call</u> and it is waiting for the execution of the corresponding <u>accept</u> statement by another task;
ACCEPTING or SELECTING :	the task is blocked as a consequence of an <u>accept</u> (<u>select</u>) statement execution and it is waiting for a corresponding <u>entry call</u>;
ENGAGED :	the task is blocked because its partner task is completing the RV (i.e. it is executing the <u>accept</u>

body);
TERMINATED : the task is terminated and it has released all its resources.

- a pointer (PA) to the record containing the actual parameter addresses after a blocking entry call;
- a set (OA) of addresses of Entries corresponding to open alternatives;
- a set (QUEUES) of queues, one for each entry declared in the task, containing the identifiers of tasks performing entry calls before the corresponding accept is executed;
- a stack (PARTNERS) containing the identifiers of the tasks involved in a RV with the task described by the GTD. It allows the handling of nested accept bodies;
- the address (DPR) of the delay part, used for timed entry call or selective wait statements;
- a link to the LTD of the same task.

Translation of the ADA Communication Constructs

The RV implementations have been coded in the sequential PASCAL language, excepting only a few functions of the lowest level coded in assembly language. The ADA tasking facilities are supported by a set of kernel primitives which can be invoked as procedures by tasks also implemented in PASCAL.

In the following it is described how the ADA parallel constructs are translated into procedural calls to kernel primitives.

- Accept statement

```
     accept  entry_name (formal parameter part)
         do
             sequence of statements
        end
```

is translated into a call to the kernel primitive:

```
procedure ACCEPT (CALLED: task_id; EN: entry_n)
```

where:

- CALLED is the identifier of the task which declares the accept statement;
- EN is the name of a PASCAL procedure implementing the accept body:

```
procedure entry-name (formal parameter part)
begin sequence of statements end.
```

- Entry call statement

```
called_task.entry_name (actual parameter part)
```

is translated into a call to a system procedure declared as:

```
procedure EC (CALLED: task_id; EN: entry_n; PA:
address);
```

where PA is a pointer to a record containing the actual parameter addresses.

- Conditional entry call

```
     select
           entry call (sequence_of_statements_1)
       else
           sequence_of_statements_2
  end select;
```

corresponds to:

```
procedure SELECTC  (CALLED: task_id; EN:
entry_n; PA: address; EP: procedure; ELSE_PART:
procedure);
```

where EP and ELSE_PART are the names of parameter-less procedures declared as:

```
procedure EP;
begin sequence_of_statements_1 end;
procedure ELSE_PART;
begin sequence_of_statements_2 end.
```

- Timed entry call

```
     select
            entry call (sequence_of_statements_
            _1)
        or
            delay statement (sequence_of_
            _statements_2)
  end select;
```

corresponds to:

```
procedure SELECTD (CALLED: task_id; EN: entry_n;
PA: address; EP: procedure; DELAY_PART:
procedure; T: time);
```

where EP and DELAY_PART are procedures containing the sequences_of statements 1 and 2 respectively.

- Selective wait

```
     select
            (when condition)
                 select alternative
            (or(when condition)
                 select alternative)
            (else
                 sequence_of_statements)
  end select;
```

where select alternative is:

```
   accept statement (sequence of statements)
 or delay statement (sequence of statements)
 or terminate;
```

is translated into a call to:

```
procedure SELECTW (SEL_MODE: mode; CALLED:
task_id; S_ALT: select_alt);
```

where:

```
type mode is: mode = (elsem, delaym, term,
                      acceptm) and
type select_alt is select_alt = array (1...n)
                     of record
                               C: function;
                              EN: entry-n;
                              EP: procedure;
                               M: mode;
                               T: time;
                              end;
```

and where:

elsem: corresponds to a select statement with the else part;
delaym: corresponds to a select statement with the delay part;
term: corresponds to a select

statement with the terminate part;
acceptm: corresponds to a select statement without else or delay or terminate parts;
C is the name of the boolean function which evaluates the guard of the i-th select alternative;
EN is the name of the procedure implementing the i-th accept body (if any);
EP is the procedure which contains the sequence of statements (if any) of the i-th select alternative;
M is the mode of the i-th select alternative;
T is the delay time when M is delaym.

More about the "Order of Arrival" Implementation

The server, procedural call and order of arrival implementations of the RV mechanism share both the translation of the ADA constructs into procedural calls to kernel primitives and the structure of the task descriptors. They only differ in some operations performed by some kernel primitives.

The order of arrival implementation, which includes the other ones to a great extent, will be described in more detail in the following. For the sake of simplicity, the ancestory relationships between tasks and their implications on task termination as well as exception handling will not be considered.

Three variables (CALLER, CALLED, RUNID) will be used to refer the tasks involved in a RV:
- CALLER contains the identifier of the task performing an entry call;
- CALLED contains the identifier of the task owning an entry;
- RUNID contains the identifier of the running task.

The ACCEPT primitive (which implements the semantics of the accept statement) firstly checks for a task in the QUEUES field of CALLED. In the case the queue corresponding to the entry is empty, both the CALLED and the RUNID states are set to ACCEPTING and the entry name is inserted in the OA field of CALLED; the next task to be executed is then selected from the ready list. If the queue corresponding to the entry in the QUEUES field of CALLED is not empty, the first task (CALLER) is extracted and its state is set to ENGAGED; the kernel procedure BEGIN_RENDEZVOUS is the invoked, with CALLER and RUNID as actual parameters.

The BEGIN RENDEZVOUS procedure refers to the formal parameters CALLER and CALLED; it firstly saves the CALLER identifier in the PARTNERS field of CALLED, then it invokes the procedure which implements the accept body. After the execution of the accept body, the control is returned to BEGIN_ _RENDEZVOUS, which will complete the RV. The RV completion part uses the information contained in the CALLED. PARTNERS stack to identity the partner of RUNID which must be resumed to the READY state. The task identifier on the top of the stack is extracted and assigned to the LCALLER variable. Since the accept body has just been executed by RUNID, if the LCALLER task is different from RUNID then it is the partner of RUNID and it will be resumed.

Otherwise, if the retrieved LCALLER task is RUNID, then three situations can occur:

1) The CALLED.PARTNERS stack is empty, i.e. a non nested accept statement occurs. As LCALLER executed the accept body, CALLED must be resumed.
2) The CALLED.PARTNERS stack is not empty, i.e. the accept body just executed by RUNID is nested inside another accept body; the former RV was initiated by a task identified as PREVIOUS_CALLER.

2a) If the PREVIOUS_CALLER state is ENGAGED then CALLED reached its accept statement after PREVIOUS_CALLER performed its entry call. In this case PREVIOUS_CALLER will be resumed by CALLED, and CALLED must be resumed by RUNID.

2b) If the PREVIOUS_CALLER state is ACCEPTING or SELECTING, then CALLED reached its accept or select statement before PREVIOUS_CALLER performed its entry call. Thus CALLED will be resumed by PREVIOUS_CALLER and RUNID must resume PREVIOUS_CALLER.

The primitive EC implements the entry call statement. If the CALLED state is ACCEPTING or SELECTING and the entry name is in the OA field of CALLED, then the CALLED state is set to ENGAGED and the procedure BEGIN-RENDEZVOUS is invoked with RUNID and CALLED as actual parameters, else the RUNID state is set to ENTRY CALLING, the pointer to the entry's actual parameter addresses is saved in the PA field of RUNID, the RUNID identifier is inserted into the queue associated with the entry name in the QUEUES field of CALLED, and the next task to be executed is selected from the ready list.

The SELECT primitive (which implements the semantics of a conditional entry call) differs from EC only when the RV is not immediately possible, in which case the procedure ELSE_PART is invoked.

The timed entry call is implemented by the procedure SELECTD: the RV is performed if and only if the corresponding accept statement is executed within a fixed time interval T, otherwise the delay part will be executed. The semantics of the timed entry call is obtained by the combined effect of the procedure SCHEDULE and of the system time scheduling mechanism implemented by the procedure PACTIVATE.

The procedure <u>SCHEDULE</u> inserts the local descriptor of RUNID in the schedule list SL, maintaining the list ordered for ascending scheduling time.

The procedure <u>PACTIVATE</u> is invoked whenever the real-time clock ticks; it controls the scheduling time of the first LTD in the SL list. If it is zero, the task descriptor is removed from the SL list and the <u>delay</u> part will not be executed. If the scheduling time equals the current clock value, that is the delay time has elapsed, the task is moved from the ENTRY CALLING to the READY state and its starting address is set to the <u>delay</u> part address.

Selective wait is the most complex construct and it is implemented by the procedure <u>SELECTW</u>. If no alternatives can be immediately selected, then the behaviour of the procedure SELECTW depends on the parameter SEL-MODE. In the case SEL-MODE is elsem, the ELSE_PART is executed. If the SEL_MODE is acceptm, i.e. there isn't an ELSE_PART, the task must wait until an open alternative can be selected. Both the RUNID and the CALLED states are set to SELECTING in analogy with the ACCEPT procedure. If the SEL_MODE is delaym then SELECTW behaves like the procedure SELECTD: the time interval, which must elapse before the DELAY PART is executed, is set to the minimum declared interval of the open delay alternatives. Finally if SEL_MODE is term, the termination of the task can be tried after setting both the RUNID and the CALLED states to SELECTING, so that the RV can be completed if the termination will not occur.

<u>Time Requirements</u>

The time requirements of the three implementations of the ADA communication primitives have been measured and compared with those of the MODOSK semaphoric primitives.

The experimental environment was composed of the MODIAC multiprocessor and of a Hewkett Packard 64000 development system.

The measurements have been performed by exploiting the real time facilities of the in-circuit emulators of the HP-64000.

It is worth noticing that the execution time of a kernel primitive is an increasing function of the number of active tasks and of the number of involved processors. Furthermore it is affected by the relative priorities of the interacting tasks when preemption occurs.

The results reported in Table 1 show the worst-case time requirements of the primitives which implement the semantics of the accept, entry call and selective wait statements, together with the worst-case requirements of the P and V operations on semaphores.

Primitive	Circumstance	Worst-case time requirements (μsec)		
		Order of arrival	Server	Procedural call
ACCEPT	Entry Call preceded Accept	1470	1470	1440
	Accept precedes Entry Call	990	990	990
EC	Entry Call precedes Accept or Sel. W.	1530	1530	1530
	Accept or Sel. W. Preceded Entry Call	1200	1850	1200
SELECTW	The Else Part is selected	1670	1670	1670
	Entry Callspreceded Selective Wait	4320	4320	3880
	Selective Wait precedes Entry Calls	2220	1970	2220
P	The semaphore is not blocking		150	
	The semaphore is blocking		750	
V	The semaphore has no waiting process		145	
	The process waiting for the longest time is inserted in the ready list		560	
	The process waiting for the longest time preempts the running process		880	

Table 1 -Time requirements of ADA and semaphoric primitives.

The quoted values are related to six interacting tasks running on three different processors.

No significant difference appears between the three implementations of rendezvous at a first glance, while the semaphoric primitives appear to perform far better. These results, however, can be misleading if one forgets that the requirements reported in Table 1 are related to single executions of the primitives and they do not take into account the number of invocations necessary for completing a rendezvous.

Only the order of arrival implementation, in fact, requires just one execution of the entry call primitive and one execution of the accept (or selective wait) primitive to complete a rendezvous.

In the server implementation, the accept (or selective wait) primitive must be invoked two times when the entry call comes last, while in the procedural call implementation the entry call primitive is executed two times when it is invoked first.

The total time requirements for a complete rendezvous are reported in Table 2.

As it was to be expected, the order of arrival implementation performs as the server one

Synchronization mode	Circumstance	WORST-CASE TIME REQUIREMENTS (μsec) order of arrival	server	procedural call
Accept Rendezvous	Entry Call precedes Accept	3000	3000	4170
	Accept precedes Entry Call	2190	4310	2190
Selective Wait Rendezvous	Entry Call precedes Select. Wait	5850	5850	6610
	Select. Wait precedes Entry Call	3420	8140	3420
P / V	V precedes P		295	
	P precedes V		1630	

Table 2. Time requirements of complete synchronizations between processes.

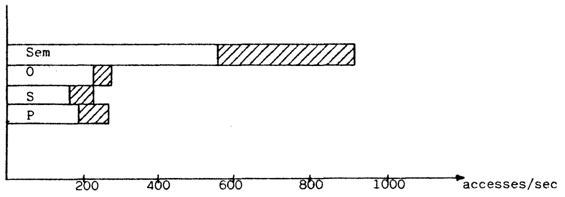


Fig. 2.1 - Slot access time = 30 μsec; Buffer size = 100

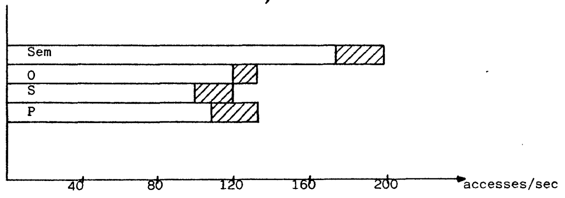


Fig. 2.2- Slot access time = 3 msec; Buffer size = 100

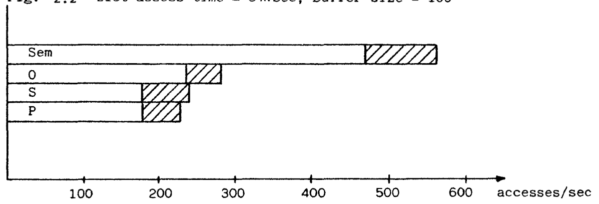


Fig. 2.3- Slot access time = 30 μsec; Buffer size = 1; Selective Wait solut.

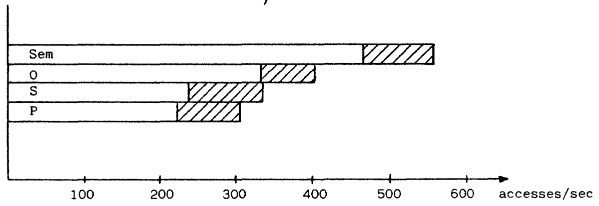


Fig. 2.4 - Slot access time = 30 μsec; Buffer size = 1; Accept solution

Fig. 2 - Access frequency to a shared buffer by producer/consumer processes

does when entry call precedes accept or selective wait, while it performs like the procedural call implementation when accept or selective wait precedes entry call. The comparison with semaphoric synchronizations is merely indicative, since the complexity, and therefore the expressive power, of the ADA tasking constructs is far greater than that of the semaphoric primitives.

In order to properly compare the performances of so different mechanisms, we must ask them for the same job in the same environment, as in the following experiment.

The classical solutions of the bounded buffer problem by means of the ADA non deterministic constructs and of the semaphoric primitives have been implemented and experimentally analyzed.

The results reported in Fig. 2.1 show the number of accesses per second to a shared buffer, where the buffer size is equal to 100 and the time required to fill or empty each buffer slot is 30 μsec.

The access frequencies range in the shaded areas, depending on the relative priorities of the involved processes.

Fig. 2.2 reports the performances of the same solutions in less extreme conditions, that is when the time for filling or empting a buffer slot is 3 msec.

The effect of reducing the buffer size to 1 (maintaining the slot access time equal to 30 μsec) is shown in the last two figures: Fig. 2.3 is related to the same solution considered in the previous experiments (using a selective wait), while Fig. 2.4 has been obtained by solving the one slot buffer problem by means of a deterministic sequence of two accept statements.

Conclusions

A general and flexible scheme for the translation of tasking constructs in ADA has been presented. Three different implementations of the rendezvous concept have been discussed, mainly from the viewpoint of efficiency. The time requirements of the three implementations have been measured and compared.

The order of arrival implementation takes, on the average, less time for completing a rendezvous than the other ones and it results to be the best solution in monoprocessor systems and in multiprocessor architectures where all the resources are shared.

In multiprocessor architectures with private resources, the fact that the access time to code and data depends on their physical allocation has to be considered.

In the server implementation, the accept body and its environment are automatically available in the context of the called task, since they are declared inside it, and therefore they can reside in the private memory of the processor which runs the called task.

In the order of arrival and procedural call implementations, this is not true since the accept body can be executed in the thread of control of the calling tasks, which possibly run on different processors.

Even if the code of accept bodies would be replicated in the private memories of every processor, the variables referred to by instructions in an accept body should be made available to all calling tasks, and therefore should be maintained in the shared memory.

The more efficient support to rendezvous of the order of arrival implementation versus the server one is unavoidably counterbalanced by a greater probability of access conflicts to the global bus.

Therefore the server model can represent the best solution in multiprocessor architectures with private resources where the global bus is the bottleneck.

In a further experiment the time requirements of the rendezvous models have been compared with those of the semaphoric model in the solution of the bounded buffer problem.

References

[1] Reference Manual for the ADA Programming Language, ANSI/MIL-STD 1815A,(Jan., 1983).

[2] J. Van den Bos, "Comments on ADA Process Communication", SIGPLAN Notices, vol.15, n. 6, (1980), pp. 77-81.

[3] A. Silberschatz, "On the Synchronization Mechanism of the ADA Language", SIGPLAN Notices, vol. 15, n. 6, (1980), pp. 96-103.

[4] E.S. Roberts, A. Evans, C.R. Morgan, and E.M. Clarke, "Task Management in ADA - A Critical Evaluation for Real-time Multiprocessor", Software Practice and Experience, vol. 11, (1981), pp. 1019-1051.

[5] J.D. Ichbiah, and others, "Rationale for the Design of the ADA Programming Language", ACM SIGPLAN Notices, vol. 14, n. 6, (1979).

[6] A.N. Habermann, and I.R. Nassi, "Efficient Implementation of ADA Tasks", Tech. Rept. CMU-CS-80-103, Carnegie Mellon Univ., (1980), pp. 22.

[7] D.R. Stevenson, "Algorithms for Translating ADA Multitasking", Proc. ACM-SIGPLAN Symposium on the ADA Programming Language, (1980), pp. 166-177.

[8] E. Falis, "Design and Implementation in ADA of a Runtime Task Supervisor", Proc. Ada TEC Conf. on ADA, Arlington, (1982), pp. 1-9.

[9] P.N. Hilfinger, "Implementation Strategies for ADA Tasking Idioms", Proc. Ada TEC Conf. on ADA, Arlington, (1982), pp. 26-30.

[10] T.P. Baker, and G.A. Riccardi, "Ada Tasking: From Semantics to Efficient Implementation", IEEE Software, (March, 1985), pp. 34-46.

[11] W. Eventoff, D. Harvey, and R.J. Price, "The Rendezvous and Monitor Concepts: Is There an Efficiency Difference?", SIGPLAN Notices, vol. 15, n. 11, (1980), pp. 156-165.

[12] S. Haridi, J.O. Bauner, and G. Svensson, "An Implementation and Empirical Evaluation of the Tasking Facilities in ADA", SIGPLAN Notices, vol. 16, n. 2, (1981), pp. 35-47.

[13] A. Jones, and A. Ardo, "Comparative Efficiency of Different Implementations of the ADA Rendezvous", Proc. Ada TEC Conf. on ADA, Arlington, (1982), pp. 212-223.

[14] S. Rivoira, and A. Serra, "A Multimicro Architecture and Its Distributed Operating System for Real-time Control", Proc. Third Conf. on Distributed Computing Systems, Miami, (1982), pp. 238-246.

[15] P. Garetti, P. Laface, and S. Rivoira, "MODOSK: A Modular Distributed Operating System Kernel for Process Control", Microprocessing and Microprogramming, vol. 9, n. 4, (1982), pp. 201-213.

[16] H.H. Lovengreen, "Parallelism in ADA", in D. Bjorner, O.N. Oest (Ed.), Towards a Formal Description of ADA, Lecture Notes in Computer Science 98, Springer-Verlag, (1980), pp. 309-432.

A BASIC PROTOCOL FOR ROUTING MESSAGES TO MIGRATING PROCESSES

T. M. Ravi and David Jefferson

UCLA Computer Science Department
University of California, Los Angeles
CA 90024

Abstract: We are investigating process migration in distributed systems with message passing. Our objective is to develop mechanisms for routing messages to and from migrating processes, that can scale up to thousands of nodes. In particular we are concerned with the optimization of storage and bandwidth required for routing messages in large systems.

In this paper we present a formal specification of a basic routing and migration protocol. The emphasis in this basic protocol is on careful modularization, synchronization, and correctness issues associated with process migration and the routing of messages to processes. We verify the basic protocol and show that it is deadlock free.

1. THE PROBLEM

In a distributed system where processes have been assigned to nodes statically, the load will vary from node to node during the course of execution. While some nodes are underutilized, others may be so heavily loaded as to become a critical performance bottleneck. Moreover some processes may be more critical than others, and the degree to which a process is critical may change from time to time as the execution proceeds. Dynamic load management is the attempt to optimize performance at run time by reallocating resources, making more available for the execution of currently critical processes.

Most current research in load management is concerned with the *policy* of process migration, i.e. deciding which process is to be moved, when, and from which source node to which destination node [NI 85, GAO 84]. Here we are concerned instead with the *mechanics* of process migration, i.e. protocols for moving a process from one node to another and delivery of messages to processes that are moving. The main technical issues arise from the fact that from the operating system's point of view both messages and processes are in motion, and messages must be routed to moving targets instead of fixed targets. In a distributed system it is impossible to predict at the time a process is to migrate whether or not there is a message in transit toward it, and thus some messages will have to be forwarded to catch up to their targets.

We are only interested in migration and routing mechanisms that can scale up to thousands of nodes. Because of this we cannot assume that a node has enough memory to keep a complete routing table mapping every process name to a node address. In our routing protocol, routing tables may have *incomplete* information, i.e. a node will generally have routing information about only some of the processes in the system. Message routing must also rely on possibly *out-of-date* routing information, i.e. new routing updates indicating that a process has moved cannot be broadcast instantaneously throughout the network.

† This work has been supported in part by an IBM Graduate Fellowship and the Jet Propulsion Laboratory Contract No. 957523.

Section 3 describes the organization and structure of the distributed system and some assumptions about the system. In Section 4 we first describe a basic routing and migration (BRM) protocol, and then give a detailed specification of the protocol. Next in Section 5 we show the absence of deadlock and prove that all messages eventually are delivered to the destination processes. Section 6 indicates some of our work in progress and future directions.

2. RELATED WORK

Process migration and forwarding of messages has been implemented in the DEMOS/MP system [Powell 83]. Our basic routing and migration protocol can be considered to be a more formal specification and generalization of their migration procedure with particular concern for modularization issues, storage issues, and synchronization issues that arise when a process has to be moved from one node to another. Our model also permits concurrent execution of several instantiations of the protocol on the same node.

Fowler [Fowler 85 & Fowler 86] proposed protocols using forwarding addresses to locate moving objects in a distributed system. In particular he analyzes the complexity of protocols for updating the address of a process that has moved, using three protocols one of which is based on the UNION-FIND path compression algorithm. This work assumes that a node has complete address tables for every process in the system. We examine mechanisms for routing when a node cannot keep a complete routing table. The main feature of Fowler's approach is that routing entries are updated only when they are along the trajectory of a message. In our protocol, we do not restrict ourselves to passive updating of routing entries, but also consider schemes where maintenance messages are propagated as a side-effect of process migration.

The Emerald system [Jul 88] has implemented mobility of process objects as well as data objects. To find an object it incorporates the forwarding address protocol proposed by Fowler. When a node is unreachable due to failure or the loss of a forwarding address, the Emerald system resorts to a system-wide broadcast to find the object.

In both Locus [Popek 85] and the Sprite operating system [Douglis 87 & Ouster 88] processes use kernel calls instead of messages for interaction with other processes. Each process is assigned a special origin site or home node to which all remote calls are sent. The special node has a forwarding address to the current location of the process to which all calls are forwarded. However this strategy has several problems. There is a residual dependency of the process on the special node even after the process has migrated several times. In large distributed systems over a period of time, the additional path length for the call could become very large compared to the actual distance to a process. When all communication to a process is required to go through a special node, process migration is no longer effective in reducing communication distances. Moreover the failure of the special node makes the process inaccessible to other processes in the system.

In the V-System [Theimer 86], when a message has to be sent, a local cache is checked for the location of the destination process. If a cache entry is found then the message is sent to the location indicated. If there is no entry for the process at the local cache, or if the process is no longer at the location indicated by the cache entry then a query is broadcast system-wide to all nodes requesting the location of the process. The local cache is updated with the response to the query and the message is forwarded to the new location.

Lu, Chen, and Liu [Lu 87] have proposed a migration mechanism that assumes that each process communicates with a finite number of processes called *adjacent* processes and each node has a table with the exact location of processes adjacent to resident processes. When a process migrates, it first blocks its adjacent processes from sending messages, migrates to a new node, sends the adjacent processes the new location of the process and re-enables the adjacent processes to send messages to the recently migrated process. This scheme ensure that there is no message for a process in transit when it migrates. In our protocol we tackle a more general problem assuming any process can communicate with any other process at any time, and that the operating system does not know which processes will communicate with one another.

Mullender and Vitanyi [Mullen 85] studied the resource location problem for a *client-server* model, in the context of the Amoeba operating system. The server posts its location to a set of nodes, and the client queries another set of nodes for the desired service. A *rendezvous* node is a node in the intersection of the two sets where the client node finds the location of the server node. They do not consider the routing of messages that may be in transit when a process moves, or that are sent based on old location information which has not yet been updated.

Scheurich and Dubois [Scheur 87] deal with the problem of the migration and location of memory pages in distributed systems. The performance of their mechanism strongly depends on the geographical, spatial, and temporal locality of page references. It uses caches for hints, and uses a broadcast search when the page is not found. Our message routing scheme basically relies on routing entries and the forwarding of messages to a process that has moved. Extensions to our skeletal protocol to be proposed elsewhere, will resemble their mechanism in that it will use caches to take advantage of localities.

Our problem is similar to the problem of routing to a mobile subscriber in packet radio systems with fixed stations [Kahn 78]. In packet radio networks, a flooding scheme is commonly used to initially locate a mobile subscriber. The weakness of this scheme, however, is the high channel bandwidth due to control messages. A modification of this scheme [Li 86] is to have a complete routing table at each station. To locate a mobile subscriber a message is sent to the last known station to which the subscriber was affiliated. If the subscriber has moved out of range of that station then a flood broadcast is initiated at the last known station.

3. THE MODEL

We envision a load managing operating system as consisting of five functional levels (Figure 1). The lowest is the *Architecture* level consisting of the nodes and the network connecting them. The *Low Level Kernel* implements reliable node-to-node message routing and manages the multiplexing of processes on each node. The next layer handles the *Mechanics of Routing and Migration* and includes protocols for process-to-process message routing, routing table maintenance, and migration of a process from one node to another. The *Load Management Policy Layer* makes decisions about how to redistribute the load. Finally

Application	cooperating processes
Load Management	migration policy
Mechanics of Routing & Migration	interprocess msg. routing migration protocol
Low Level Kernel	internode msg. routing process multiplexing
Architecture Level	many processors MIMD no shared memory

Figure 1 : Functional Levels in System

the *Application* layer consists of a collection of cooperating processes that communicate asynchronously by messages.

Each level of the system is assumed to be reliable. Communication between nodes is by asynchronous messages, takes finite time, and need not be order preserving. Each node may have multiple processes resident on it. A process may send a message to any other process at any time, addressing it by name. In this paper we are interested in the implementation of the third layer, handling the *Mechanics of Routing and Migration*. Message routing between nodes is handled by the *Lower Level Kernel*, and we will not be concerned with how it is done.

4. THE BASIC PROTOCOL

We now present a Basic Routing and Migration protocol, BRM, which forms a basis for the subsequent development of more complex protocols that optimize performance. The BRM is designed with careful attention to modularization to allow for different possible implementations of sub-modules and so that different features can be added on to the basic protocol for performance enhancement. This protocol assumes that buffers of unlimited size are available at each node to store messages while remote routing entries are being obtained.

Associated with each process, the operating system maintains a *migration-count* α which is incremented whenever the process migrates. Each node maintains a possibly incomplete *routing table* containing *entries* that among other things map process names to node addresses. Routing entries for a process are marked with a migration-count field that is equal to the migration-count of the process when it was located at the node indicated by the routing entry. The *status* of a routing entry indicates whether it is *permanent* or *temporary*, i.e. whether it remains in existence for the lifetime of the process or not. The *updates* field indicates the number of pending updates or update requests to remote nodes from the node where the entry is located. Before an update to remote nodes is started this field is incremented by one and when the update is complete it is decremented by one. A temporary routing entry can be deleted only when the value of this field is zero. A *lock* field controls access to the routing entry. The structure of a routing entry is shown in Figure 2. When a process becomes resident on a node, a new routing entry for that process is created at that node if one does not already exist.

Process Name	Lock	Location
Migration-count	Status	Updates

Figure 2 : Routing Entry

When a process is created, routing entries for the process are distributed to a subset of the nodes in the system and remain in existence for the lifetime of the process. It is not necessary that every node have a routing entry for every process. However each node *U* has access to a set ***&surrogate(U, q)*** of nearby nodes that have a routing entry for a process *q*. *U* itself is not considered a member of ***&surrogate(U, q)***. The set ***&surrogate(U, q)*** does not have to contain all the nodes in the system that have a routing entry for *q*. If *U* has a permanent routing entry for *q* then ***&surrogate(U, q)*** may be an empty set. However if *U* does not have a permanent routing entry for *q* then ***&surrogate(U, q)*** must be nonempty.

In brief the protocol for routing a message to a process is as follows :

a. When node *U* must route a message to process *q*, either at the request of process *p* on *U* or in order to forward a message addressed to *q*, the operating system first checks if a routing entry for *q* exists at node *U*. If so the message is either *delivered* locally or *routed* to the proper node depending on whether the routing entry points to *U* or not. However if no routing entry for the process is found at *U* then a *routing fault* occurs.

b. When a *routing fault* for a message addressed to *q* occurs at node *U*, the message is buffered at *U* and a request is sent to one or more surrogates of *U* which have routing entries for *q*, i.e. nodes in ***&surrogate(U, q)***. When a routing entry for *q* is received in response, the message is routed to the node indicated by the routing entry.

c. If a node *V* receives a message for a process *q* that is not resident on *V*, it *re-routes* the message. The procedure for *re-routing* a message from *V* is identical to that for routing a message originating at that node.

Note that in some implementations of the BRM protocol, it is possible to receive several replies to a request for a particular routing entry from the surrogates. The first routing entry that is received is used to route the waiting message. If a routing entry is received and there are no pending messages then it is ignored.

The migration protocol for moving process *q* from node *U* to *V* is informally described below. Process migration is atomic with respect to message routing, i.e. a message cannot be delivered to a process while it is migrating.

d. The temporary or permanent routing entry for *q* at node *U* is locked, and the execution of *q* is stopped at node *U*. Node *U* sends a MOVE system message, containing the process state (and program code if necessary) to node *V*.

e. When node *V* receives the MOVE message, the migration-count of *q* is incremented, *q* is installed and restarted at *V*, and the routing entry for *q* at *V* is modified to indicate its current location and migration-count. If there is no routing entry for *q* at *V* then one is included in the routing table at *V*, marked as temporary, and assigned the new location and migration-count of *q*. A system message MOVE_CONFIRM is sent back to node *U* indicating that *q* is now located at *V*.

f. Node *U* on receiving MOVE_CONFIRM, modifies the routing entry for *q* to indicate that *q* is at *V*, marks it with the incremented migration-count and unlocks the routing entry. Node *U* sends a routing update message to all surrogate nodes of *U* with routing entries for *q*.

Figure 3 illustrates the migration protocol. The choice of the surrogates and the method for the access and update of remote routing entries are left open in the BRM protocol. Obtaining a remote routing entry can be done in a number of ways such as bounded broadcasts, hashing etc., which we will not go into here.

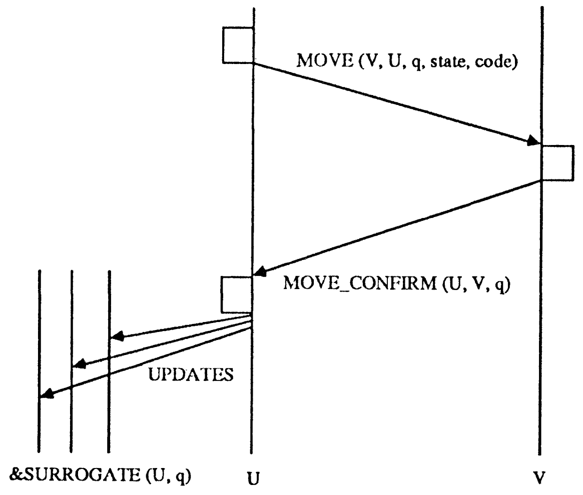


Figure 3 : Migration Protocol

In the basic protocol temporary routing entries for a process are created at a node for two different reasons. If a process *q* is resident at a node *U* which does not normally have a routing entry for *q*, then a temporary routing entry for *q* is created at *U* to indicate that *q* is located at *U*. Another instance where a temporary routing entry is used is to facilitate the efficient atomic update of routing tables of surrogates. In order to reduce the synchronization delay during which a routing entry at *U* is unavailable, an atomic update can be achieved by creating a temporary routing entry at *U*, and sending asynchronous updates to the remote nodes that have to be updated. Creation of temporary routing entries can however lead to an increase in storage over time and hence these latter temporary entries must be deleted when the update is completed.

For correctness, the routing table update procedure has to satisfy the following specifications :

1. When a routing table update is received at a node *W* indicating that process *q* is located at *V* with migration-count α, then the routing entry for *q* at *W*, if any, is updated only if α is greater

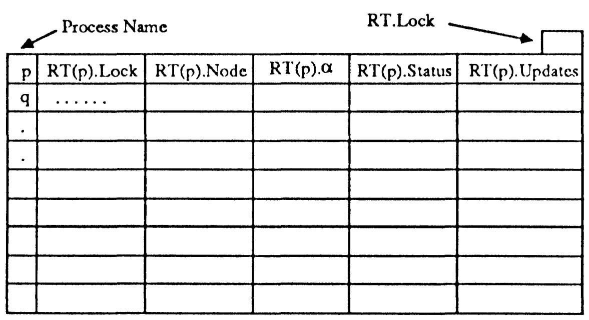


Figure 4 : Routing Table (RT)

than the migration-count of the routing entry for *q*.

2. The set of remote routing entries for *q* that can be queried when a routing-fault occurs at node *U*, must be a subset of those updated by *U* when *q* migrates away from *U*.

3. Any temporary entry for *q* at *U* can be deleted only after all members of *&surrogate(U, q)* have been updated.

If a process *q* migrates away from *U*, returns and migrates again, then it is possible that a new request for updating surrogate nodes of *U* with routing entries for *q* can be generated while a previous remote update for *q* is ongoing. The operating system therefore has to keep track of the number of requests for remote updates that are not yet complete. Only when all remote updates have completed can a temporary routing entry for *q* at node *U* be deleted.

Access to the tables is controlled by locks to ensure that possible concurrent access to a table at a node is synchronized. We design for the possibility that several instantiations of the BRM protocol may be running simultaneously at a node responding to different routing and migration activities. Associated with each table are two levels of locks, one lock to control insertion and deletion of entries from the table, and the other associated with each entry, to control read or write access to that entry.

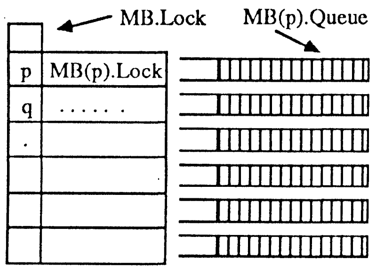


Figure 5 : Message Buffer Table (MB)

Figures 4 and 5 show the routing table (RT) and the message buffer table (MB). We refer to these tables at node *U* as RT_U and MB_U, but we drop the subscripts when the location of the table is clear from the context. Each entry in MB has a message queue to store messages. We will always assume that the buffer at node *U* is large enough to store those messages destined for a process *q* that arrive at *U* while *q* is migrating from *U*. It is a flow-control problem to guarantee that this assumption is true, but we do not deal with this issue in this paper. Enqueue and Dequeue operations insert and remove messages from the queue. The Empty operation checks to see if a queue is empty. Lock (A.Lock) and Unlock (A.Lock) obtain and release A.Lock. A lock that is requested and is unavailable is retried and the algorithm does not proceed to the next step till the lock is obtained. Insert&Lock creates an entry that is initialized as locked. Delete unlocks an entry and then removes it.

Send is a kernel call to route a message to the node indicated in the message. **When** indicates the action to be taken on the receipt of a message from the kernel or a request from the higher levels of the operating system.

The algorithm for the routing and migration protocol follows. Each step of the algorithm is invoked asynchronously upon the arrival of a message or on receiving a command from the load management or application level.

BASIC ROUTING & MIGRATION PROTOCOL

Variable for each Process q

q.α	migration-count for process q. Incremented whenever q moves.

Variables at each Node U

RT	routing table located at node U containing routing entries.
RT.Lock	lock associated with table RT that has to be obtained before any routing entries are inserted or deleted from RT.
RT(q)	routing entry for process q. By convention we say it is equal to Nil if there is no entry for q in the routing table at U. RT(q) is deleted from RT by the execution of Delete(q, RT). RT(q) is created by executing Insert&Lock(q, RT, RT(q).Lock), that creates an entry that is already locked.
RT(q).Node	possibly out-of-date node address for q.
RT(q).α	migration-count associated with the above routing entry.
RT(q).Status	indication of whether this routing entry is permanent or temporary. Permanent entries are marked "permanent" and temporary entries are marked "temporary".
RT(q).Updates	number of updates to remote

	routing entries whose completion has not yet been confirmed.
RT(q).Lock	lock for routing entry that has to be obtained before the entry is read or written into.
MB	message buffer table at node U consisting of a list of queues of messages destined for different processes and waiting for the resolution of routing-faults.
MB.Lock	lock associated with MB that has to be obtained before MB(q) can be created or removed from MB.
MB(q)	message buffer entry to indicate the existence of a queue of user messages whose destination is q. MB(q) = Nil if there are no messages whose destination is q. MB(q) is deleted from MB by the execution of Delete(q, MB). MB(q) is created by executing Insert&Lock(MB(q), MB(q).Lock), that creates a entry consisting of an empty queue that is already locked.
MB(q).Queue	queue of user messages that are waiting for the resolution of routing-faults and whose destination is q.
MB(q).Lock	lock for MB(q) that has to be obtained before a message can be enqueued or dequeued from MB(q).

Messages Sent & Received at each Node U

SMSG(U, q, text) system message sent to node U containing user message text for process q.

MOVE(U, V, q, state, code) system message for the migration of q from node V to U, containing the process state and code.

MOVE_CONFIRM(V, U, q) system message in response to a MOVE message; sent from node U to V indicating that process q is executing on U.

RE(U, q, W) system message sent to U indicating that q is believed to be at W.

UPDATE_CONFIRM(U, q) system message to confirm that updates for q sent to remote nodes have been received. It may be a message sent by U to itself.

Initialization Initially the migration-counts of all processes are zero and routing entries for each process are set up in the routing tables of some subset of nodes. The routing entries for a process consist of a node address which is the initial location of the process, and the entries are marked with a migration-count of zero, updates equal to zero and status permanent.

Procedures & Functions

Function ROUTE_LOCAL(q, text)

{ Attempts to deliver or route messages based on the local routing table. Delivers the message destined for q if q is located at node U. Else if there is a routing entry for process q at U then the message is forwarded to the node location given by the routing entry. }

```
begin
    Lock(RT.Lock)
    if RT(q) ≠ Nil then begin
                              {routing entry found}
        Lock(RT(q).Lock)
        Unlock(RT.Lock)
        if RT(q).Node = U then begin
                              {locally delivered}
            Place text in message queue
               for process q
            result := delivered
        end else begin
                              {re-routed}
            send SMSG(RT(q).Node, q, text)
            result := re_routed
        end
        Unlock(RT(q).Lock)
    end else begin
        Unlock(RT.Lock)
        result := failed
    end
    return(result)
end ROUTE_LOCAL
```

Procedure ROUTE_REMOTE(q, text)

{ Buffers the content of a message in the waiting table and requests for a remote routing entry for the destination process. The message with content text destined for q and is buffered in the waiting table at node U and a remote routing entry for q is requested. }

```
begin
    Lock(MB.Lock)
    if MB(q) = Nil then begin
            {no message buffer entry for q}
        Insert&Lock(q, MB, MB(q).Lock)
        Unlock(MB.Lock)
            {buffer message}
        Enqueue((q, text), MB(q).Queue)
        Unlock(MB(q).Lock)
```

```
            {request remote routing entry}
        REQ_REMOTE_RE(q)
    end else begin
            {other messages for q waiting}
        Lock(MB(q).Lock)
        Unlock(MB.Lock)
            {buffer message}
        Enqueue((q, text), MB(q).Queue)
        Unlock(MB(q).Lock)
    end
end ROUTE_REMOTE
```

Procedure DELETE_RE(q)

{ Delete the routing entry for q from the local routing table if the entry is temporary, all updates to remote nodes have completed and the process is not resident at U. }

```
begin
    Lock(RT.Lock)
    if RT(q) ≠ Nil then begin
        Lock(RT(q).Lock)
        if RT(q).Status = temporary
            {temporary routing entry}
        and RT(q).Node ≠ U
            {process has not moved back}
        and RT(q).Updates = 0 then begin
            {remote updates complete }
            Delete(q, RT)
            {delete temporary routing entry}
        end else begin
            Unlock(RT(q).Lock)
        end
    end
    Unlock(RT.Lock)
end DELETE_RE
```

Procedure REQ_REMOTE_RE (q)

{ This is a procedure to obtain a remotely located routing entry for q. In the Basic Routing & Migration Protocol the implementation of REQ_REMOTE_RE is left unspecified, as it can be done in a variety of ways. REQ_REMOTE_RE (q) executed at different nodes may return different values.

The set of remote routing entries that are queried are a subset of those updated.

In response to messages sent in this procedure, node U receives a message or messages RE(U, q, V) indicating that q is believed to be at V. }

Procedure UPDATE_REMOTE_RE (q, V, α)

{ Sends update messages to some subset of remote routing entries from U. The routing entries are updated to indicate that process q is located at node V with migration-count α, but only if α is greater than the migration-count of the routing entry being updated. The implementation of UPDATE_REMOTE_RE is left unspecified. The subset contains at least those remote routing entries that can be queried from U.

If U has a temporary routing entry for q then in response to update messages sent in this procedure node U receives a UPDATE_CONFIRM (U, q) message indicating that all nodes in the subset have received the update. }

Algorithm at each Node U

```
<1> when a process on node U requests sending
          of a message to process q begin
            {call from application layer}
        result := ROUTE_LOCAL (q, text)
        if result = failed then begin
            ROUTE_REMOTE (q, text)
        end
    end

<2> when SMSG(U, q, text) arrives begin
            {message from outside arrives}
        result := ROUTE_LOCAL (q, text)
        if result = failed then begin
            ROUTE_REMOTE (q, text)
        end
    end

<3> when RE(U, q, V) arrives  begin
        Lock(MB.Lock)
        if MB(q) ≠ Nil then begin
            {messages for q in buffer}
            Lock(MB(q).Lock)
            while not Empty (MB(q).Queue)
                                        do begin
                (q, text) := Dequeue(MB(q).Queue)
            {re-route messages in buffer}
                send SMSG(V, q, text)
            end
            Delete(q, MB)
            Unlock(MB.Lock)
        end else begin
            {no messages for q in buffer}
            Unlock(MB.Lock)
        end
    end

<4> when policy software requests
       migration of q from U to W  begin
            {call from load management layer}
        Lock(RT (q).Lock)
            {block access to routing entry}
        Remove q from execution
        send MOVE(W, U, q, state, code)
    end

<5> when MOVE(U, V, q, state, code)
                                  arrives begin
        q.α := q.α + 1
        Install process q with its state
        and code at node U
      Lock(RT.Lock)
      if RT(q) = Nil then begin
          {create temporary routing entry}
          Insert&Lock(q, RT, RT(q).Lock)
          Unlock(RT.Lock)
          RT(q).Status := temporary
          RT(q).Updates := 0
          RT(q).Node := U
          RT(q).α := q.α
          Unlock(RT (q).Lock)
      end else begin
          {permanent routing entry exists}
          Lock(RT (q).Lock)
          Unlock(RT.Lock)
          {update routing entry}
          RT(q).Node := U
          RT(q).α := q.α
          Unlock(RT (q).Lock)
      end
      send MOVE_CONFIRM(V, U, q)
  end

<6> when MOVE_CONFIRM(U, W, q) arrives begin
          {update local routing entry}
      RT(q).Node := W
      RT(q).α := RT(q).α + 1
          {pending update to remote entries}
      RT(q).Updates := RT(q).Updates + 1
      α := RT(q).α
          {unblock access to routing entry}
      Unlock(RT(q).Lock)
          {update remote routing entries}
      UPDATE_REMOTE_RE(q, W, α)
  end

<7> when UPDATE_CONFIRM(U, q) arrives begin
          {a remote update is complete}
      Lock(RT.Lock)
      if RT(q) ≠ Nil then begin
          Lock(RT(q).Lock)
          Unlock(RT.Lock)
          RT(q).Updates := RT(q).Updates - 1
          Unlock(RT(q).Lock)
          {attempt to delete entry}
          DELETE_RE(q)
      end else begin
          Unlock(RT.Lock)
      end
  end
```

5. CORRECTNESS ARGUMENT

In this section we discuss the validation of the basic protocol by showing the absence of deadlocks and by proving that under appropriate conditions messages that are routed according to the protocol are eventually delivered to the destination process.

We first examine the possibility of local and distributed deadlock in the BRM protocol. Every step of the protocol has the following properties.

(i) An attempt is made to obtain a lock for a table entry only when the entry exists and the entry is guarded from deletion while the protocol is waiting for its lock.

(ii) The actions in each step other than waiting for locks take finite time.

Every step except <6> has the additional property.

(iii) In each step table entries are locked before they are accessed.

Every step except <4> has the additional property.

(iv) Before completion of a step all locks obtained in the step are released.

By assumption we have the following property.

(v) Low level node-to-node message delivery takes finite time.

There are two levels of locks, one associated with an entry that has to be acquired before reading or writing to the entry, and the other associated with an entire table to control insertion, deletion and testing the existence of an entry. An entry lock and the corresponding table lock on the same node can be viewed as a *parent-child* lock pair. Obtaining the table lock does not prevent concurrent acquisition of an entry lock belonging to the table.

In the BRM protocol, the procedure for creating an entry is first to lock the table, test for the existence of the entry in the table, and if it doesn't exist then execute an Insert&Lock operation that creates an entry that is already locked. The table lock can now be released, the entry written into and the entry lock released. As the table lock has to be obtained to create an entry, two entries for a table cannot be created concurrently and there is no danger of creating duplicate entries.

An entry is deleted by first locking the table, then locking the entry and then executing a Delete operation that removes the entry. The table is then unlocked. As the entry has to be locked before it can be deleted hence in the BRM protocol it is not possible to delete an entry that is simultaneously being accessed.

To access an entry only the entry lock has to be obtained. However, in the protocol whenever there is a possibility that an entry does not exist, the table is first locked and the existence of the entry in the table is checked. If the entry is in the table the entry is locked and the table lock is released. Hence in the BRM protocol there is no possibility of waiting forever in attempting to lock an entry that does not exist. Thus property <i> holds for all steps of the protocol.

A *causal path* is a sequence of dependent steps or events that can be on the same node or across nodes. Steps <4>, <5> and <6> form part of a causal path invoked when a process *q* migrates say from *U* to *V*, with steps <4> and <6> executing on node *U* and step <5> executing on *V*. The sequence of steps <4>, <5> and <6> take finite time because of properties (ii) and (v). The lock $RT_U(q).Lock$ obtained in step <4> is not released at the end of the step. Subsequently when step <6> is invoked at *U* the routing entry for *q* can be accessed because the lock is already obtained by an earlier step in the causal path. Finally in step <6> the lock $RT_U(q).Lock$ is released. We observe that even though properties (iii) and (iv) do not hold for all steps, for every causal path through the protocol entries are locked before they are accessed, and all locks obtained in the course of the path are released before the path terminates.

Theorem The Basic Routing and Migration Protocol is deadlock free.

Proof: When a causal path needs only one lock, it is obtained and released after finite time. For all causal paths in our protocol except the sequence of steps <4>, <5> and <6> invoked when a process migrates, at most two locks are held at any time. When two locks are held at a time, the two locks are always an entry lock and the corresponding parent table lock and they are always obtained monotonically - first the parent table lock and then the entry lock. The concurrent execution of two paths of the protocol cannot result in a deadlock because the pair of locks have to be obtained in order. Moreover as only a corresponding parent-child lock pair can be held at the same time and not an entry-entry lock pair hence there is no possibility that three or more concurrent executions of the protocol can lead to a deadlock cycle. Thus there can be no deadlock when causal paths in the protocol other than the path related to migrating a process are executed concurrently on the same node or on different nodes.

Now consider the causal path that results when a process migrates. We assume that the Load Management level of the operating system is *well behaved* with the following properties:

i. It does not request a process to be migrated from a node to itself.

ii. It does not request a process to be moved from a node if it is not located at the node.

Because of assumption (i), steps <4> and <6>, and <5> that belong to a causal path cannot be invoked on the same node. Let us consider the migration of a process q from U to V. Steps <4> and <6> are invoked on U and <5> is invoked on V. A routing entry for q at U and a parent-child pair of the routing table and routing entry at V can be locked at the same time. Because the pair of locks at V is obtained in parent-child order hence a causal path with steps <4> , <5> and <6> cannot be involved in a deadlock with any other causal path. From assumption (ii) and the fact that every process has an unique name, simultaneous migrations from a node or between a pair of nodes has to involve locking routing entries for different processes which can not result in deadlock. Hence any two causal paths of the protocol can not be involved in a deadlock.

We have thus shown that the protocol is deadlock free. ■

Figure 6 illustrates the absence of deadlock in the scenario when two processes q and p migrate simultaneously from U to V and V to U respectively.

Before we proceed with the verification of the basic protocol we introduce a few definitions. A *process trajectory* at time t is defined as the sequence of node locations where the process has been resident up to time t. Since initialization, if a process q has been resident on nodes X_0, X_1, ... and at time t is resident at X_η, then the process trajectory is $trajectory(q, t) = X_0 X_1 X_2 \cdots X_\eta$. If the process q moves from X_η to $X_{\eta+1}$ at time t', where $t' > t$ then $trajectory(q, t') = X_0 X_1 X_2 X_3 \cdots X_\eta X_{\eta+1}$.

Similarly let a *message trajectory* at time t for a message be the sequence of node locations to which the message has been routed up to time t. If a message M_q whose destination is process q, originates at node Y_0 and is initially routed to Y_1, then re-routed to Y_2, Y_3, ... and is last routed to Y_τ, then the message trajectory at time t is $trajectory(M_q, t) = Y_1 Y_2 Y_3 \cdots Y_\tau$. For technical reasons we leave the origin of the message Y_0 out of the message trajectory. When the message is re-routed from node Y_τ to $Y_{\tau+1}$ then the old message trajectory is appended with $Y_{\tau+1}$ to obtain the new message trajectory.

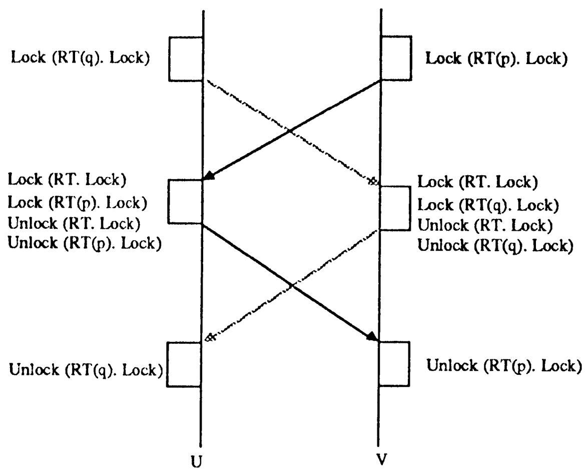


Figure 6 : Simultaneous process migration between U and V

We define the functions *delegate* and *toward*. Let q be a process and U and X be nodes. The nondeterministic function *surrogate* was earlier defined as:

$surrogate(U, q) =$ Some node X such that $X \neq U$ and $RT_X(q) \neq nil$

The function *surrogate* need not return the same node value every time it is invoked. We define *&surrogate* (U, q) as the set of all possible nodes that can be returned by *surrogate* (U, q).

The node with the routing entry used to route messages to process q from node U is given by function *delegate*(U, q).

$delegate(U, q) =$ If $RT_U(q) \neq nil$ then U
else $surrogate(U, q)$

The function *delegate*(U, q) can never be *nil*. In accordance with procedure ROUTE_MSG, if there is an entry for q at U then it is chosen for routing messages for q at node U, else an entry for q located at *surrogate* (U, q) is used.

The function *toward*(U, q) computes the address of a node to which messages for a process q are routed (or re-routed) from node U, i.e. the node that *delegate*(U, q) believes is the location of q.

$toward(U, q) = RT_{delegate(U,q)}(q).Node$

In order to show the correctness of the Basic Routing and Migration Protocol we prove the following theorems:
First we prove that while routing entries are not necessarily up to date, they always point to a past or the present location of the process and have a migration-count equal to the number of the times the process had (has) migrated when it was (is) at that location.

Theorem 1 Let the process trajectory for process q at time t be $trajectory(q, t) = X_0 X_1 \cdots X_\gamma \cdots X_\eta$. For any node U at any time $RT_U(q)$ is unlocked, if $\gamma = RT_U(q).\alpha$ then $RT_U(q).Node = X_\gamma$.

Proof: Initially, all routing entries for q point to the original location of

q, i.e. X_0, and all routing entries for q have migration-count zero. Also initially a process has a migration-count equal to zero. A routing entry can be modified only in step <6> and by the receipt of updates. Steps <4> to <6> and the remote update procedure are invoked whenever a process migrates to a new location. Every time a process migrates, its migration-count is incremented (step <5>). We observe in steps <6> and from the specifications of the update procedure that the routing entries that are modified when a process migrates to its γ+1th location, point to the new location of the process and are assigned a migration-count γ. Thus any routing entry for a process always points to the present or past locations of the process, and a routing entry with migration-count γ points to the γ+1th location of the process. ■

We now show that messages to a process from a node that is a previous location of the process, will be routed to a more recent location of the process.

Theorem 2 At any time after a process q has left its μth location U, we have $RT_{delegate(U,\ q)}(q).\alpha \geq \mu$.

Proof: Let us assume that the process q with migration count $\mu-1$ resides on node U which is the μth location of q. If process q moves from U to V, the routing entry for q at U if non null is updated to point to V and it is assigned the latest migration-count of q, i.e. μ. If the routing entry for q at U is null, then all nodes belonging to *&surrogate*(U, q), are updated (<6> and specification of remote update procedure) to point to V and assigned the migration-count μ. During the update of nodes *surrogate*(U, q), the routing entries for q at *surrogate*(U, q) are inaccessible from U because a temporary local routing entry for q is created at U and remains in place till all the surrogates have been updated (<6> & <7>). Thus at the moment process q completes its migration away from its μth location U, a routing entry for q at all values of *delegate*(U, q) will have a migration-count μ.

We now show that the migration-count field of a given routing entry on a given node never decreases. From the update procedure which describes the action taken by a node on receiving an update message, we observe that the routing entry for a process q is left unchanged unless the migration-count of the update is strictly greater than the migration-count of the routing entry. If the process q revisits U then the migration-count of its routing entry at U can only increase when q leaves U because the migration-count of a process is monotonically increasing on every move. Therefore the routing entry for q at any *delegate*(U, q) will have migration-count $\geq \mu$, where U is the μth location of q. ■

In Theorem 3 we prove that if messages for a process are routed to a new node from another node then either the process is resident on the new node or else the new node has a more recent routing entry for the process.

Theorem 3 At all times, either $RT_{toward(U,\ q)}(q).Node$ = $toward(U,\ q)$ or $RT_{delegate(toward(U,\ q),\ q)}(q).\alpha > RT_{delegate(U,\ q)}(q).\alpha$.

Proof: Let $\gamma = RT_{delegate(U,\ q)}(q).\alpha$. From Theorem 1 we know that any routing entry for process q with migration-count γ points to the γ+1th node in the process trajectory of q. Therefore *toward*(U, q) is the γ+1th node in the process trajectory of q. Suppose q is not at node *toward*(U, q), i.e. $RT_{toward(U,\ q)}(q).Node \neq toward(U,\ q)$. From Theorem 2 we know that if node *toward*(U, q) is the γ+1th location of q, then $RT_{delegate(toward(U,\ q),\ q)}(q).\alpha \geq \gamma+1$.

Therefore $RT_{delegate(toward(U,\ q),\ q)}(q).\alpha > RT_{delegate(U,\ q)}(q).\alpha$. ■

Finally, we show that a message to a process follows the trajectory taken by the process.

Theorem 4 The trajectory of a message is a subsequence (not necessarily contiguous) of the trajectory of its target process.

Proof: From the definition of the function *delegate* we know that either a node U has a routing entry for q, or else *surrogate*(U, q) has a routing entry for q. From procedure ROUTE_MSG we observe that messages for a process q are routed from any node U to *toward*(U, q). If a message is routed to a node *toward*(U, q) where the process is not resident, then the message is re-routed to *toward*(*toward*(U, q),q). Suppose the message is routed from say Y_0 to Y_1, from Y_1 to Y_2 and so on, and arrives at Y_τ at time t. At time t, the message trajectory is $trajectory(M_q,\ t) = Y_1\ Y_2\ Y_3 \cdots Y_\tau$.

Let the process trajectory be $trajectory(q,\ t)$ = $X_0\ X_1 \cdots X_\gamma \cdots X_\eta$. From Theorem 1 we know that any routing entry for a process points to a node on the process trajectory. Therefore at any time t the jth node in $trajectory(q,\ t)$ maps to the μth node of $trajectory(M_q,\ t)$ such that $X_{j+1} = Y_\mu$. From Theorem 3 we know that this mapping is strictly monotonic i.e. if $X_{j+1} = Y_\mu$ and $X_{\gamma+1} = Y_\gamma$ and if $j > \gamma$ then we have $\mu > \gamma$.

Therefore a message for a process follows a trajectory that is a subsequence of the trajectory taken by its target in such a way that every time a message is re-routed it is sent to a more recent location of the process. ■

We now show that under certain conditions for the velocity of messages and processes the BRM protocol does correctly deliver all messages to their target processes. The message velocity is the rate of increase of the number of nodes in the message trajectory and incorporates the time that a message is waiting for the resolution of routing faults.

Proposition If a message always travels a shortest physical distance route between stops along its trajectory, and if its average velocity is strictly greater than the average velocity of its target process, then the message will be delivered to its target.

Proof: By Theorem 4 the trajectory of the message is a subsequence of the trajectory of the target process. By the triangle inequality the message travels a physical distance between any two stops along its trajectory that is less than or equal to the physical distance the process traveled between the same two stops. Hence, if the average velocity of a message is higher, and the average distance traveled between points is equal or shorter, the message will *catch up with* the process. ■

Figure 7 illustrates a message from process p following the trajectory of process q. The migration-count of q is originally zero and every time q migrates, its migration-count is incremented by one. The message is delivered to q when the migration-count of q is ten.

6. WORK IN PROGRESS

At UCLA, we are engaged in designing and implementing a distributed load management operating system on a 32 node iPSC hypercube. For this relatively small system, we have complete routing tables at each node. The BRM protocol is extended so that whenever a process migrates, an asynchronous update is broadcast to all the nodes.

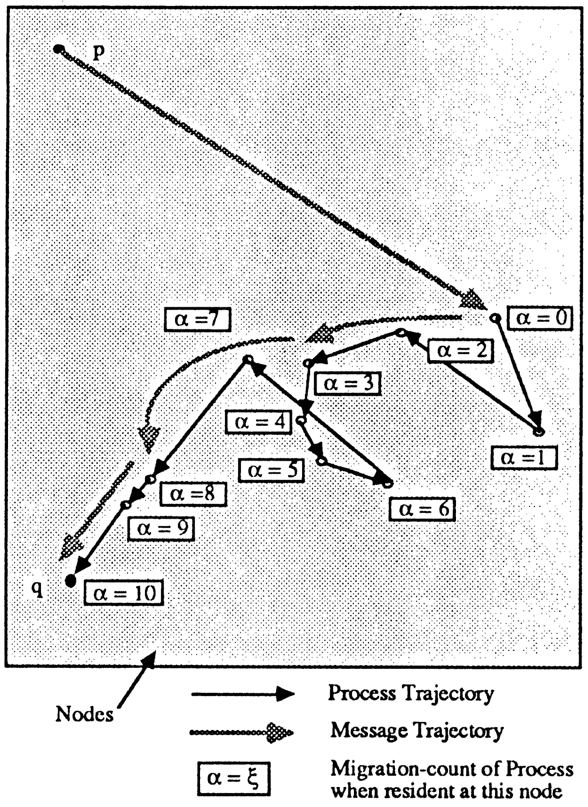


Figure 7 : Route taken by a message from process *p* to *q*

For large systems, we have developed extensions to the BRM protocol that take advantage of localities. In particular we have developed an extension called the hierarchical protocol where routing entries for a process are updated with non-uniform frequency. We have proposed a scheme for the addition of caches to the BRM protocol to take advantage of temporal locality in communication . We are currently developing analytical and simulation models to compare the bandwidth, memory requirements, delay and reliability properties of these extended protocols.

REFERENCES

Douglis 87 Douglis, F., and J. K. Ousterhout, "Process Migration in the Sprite Operating System", *Seventh International Conference on Distributed Computing*, Sept. 1987, pp. 18-25.

Fowler 85 Fowler, Robert J., "Decentralized Object Finding Using Forwarding Addresses", PhD thesis, University of Washington, Seattle, Washington, December 1985. (Department of Computer Science Technical Report TR85-12-1)

Fowler 86 Fowler, Robert J., "The Complexity of · Using Forwarding Addresses for Decentralized Object Finding", *Proceedings Fifth ACM Symposium on the Principles of Distributed Computation*, Calgary, Canada, August 1986.

Gao 84 Gao, Chuanshan, Jane W. S. Liu, and Malcolm Railey, "Load Balancing Algorithms in Homogeneous Distributed Systems", *Proc. 1984 Int. Conf. Parallel Processing*, August 1984.

Jul 88 Jul, Eric, H. Levy, N. Hutchinson and A. Black, "Fine-Grained Mobility in the Emerald System", *ACM Transactions on Computer Systems*, Vol. 6, No. 1, Feb. 1988.

Kahn 79 Kahn, Robert E., S. A. Gronemeyer, J. Burchfiel and R. C. Kunzelman, "Advances in Packet Radio Technology", *Proceedings of the IEEE* , Vol. 66, No. 11, pp. 1468-1496, Oct. 1978.

Li 86 Li, Victor O. K., and Rong-Feng Chang, "Proposed Routing Algorithms for the US Army Mobile Subscriber Equipment (MSE) Network", *IEEE MILCOM 86*, Monterey, California, October, 1986.

Lu 87 Lu, Chin, Arthur Chen and Jane W. S. Liu, "Protocols for Reliable Process Migration", *IEEE Infocom 87*, San Francisco, California, March - April, 1987.

Mullen 85 Mullender, Sape J. and Paul M. B. Vitanyi, "Distributed Match-Making for Processes in Computer Networks", *Proceedings Fourth ACM Symposium on the Principles of Distributed Computation*, Minacki, Canada, August 1985.

Ni 85 Ni, Lionel M., Chong-Wei Xu and Thomas B. Gendreau, "A Distributed Drafting Algorithm for Load Balancing", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 10, October 1985.

Ouster 88 Ousterhout, J. K., et al., "The Sprite Network Operating System", *Computer*, February 1988.

Popek 85 Popek, G. J. and B. J. Walker, "The LOCUS Distributed System Architecture", Computer Systems Series, The MIT Press, 1985.

Powell 83 Powell, M. L. and B. P. Miller, "Process Migration in DEMOS/MP", *Proc. 9th Symposium Operating Sys. Principles*, October 1983.

Scheur 87 Scheurich, Christoph and Michel Dubois, "Dynamic Memory Allocation in a Mesh-Connected Multiprocessor", *Proc. of the 12th Annual Hawaii International Conference on System Sciences*, January 1987.

Theimer 85 Theimer, M., "Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems", Stanford University Technical Report STAN-CS-86-1128, June 1986.

AN OPTIMAL SCHEME FOR DISSEMINATING INFORMATION

Yijie Han and Raphael Finkel
Computer Science Department
Patterson Office Tower
University of Kentucky
Lexington, KY 40506-0027

Abstract — We present an optimal communication pattern that achieves complete dissemination of information among n machines in $\lceil \log n \rceil$ time by repeated use of point-to-point messages. This scheme improves previous results in this field by removing restrictions on n (it need not be a power of 2) and by introducing reorganization to tolerate single-machine faults.

Introduction

Data broadcasting is a very important operation in parallel and distributed systems [1,2,3,4,5]. Alon *et al.* [1] have characterized the data-dissemination process as one in which each machine repeatedly receives, modifies, and forwards messages. All machines act in synchrony; in each round, each machine sends and receives at most one message. Alon *et al.* have shown that broadcasting can be accomplished by data dissemination without physical broadcast.

The fundamental mechanism of Drezner [2] and Alon *et al.* [1] disseminates information by repeatedly doubling the number of machines that receive the information during each round. Information originally in machine i_1 can be sent in one round to machine i_2. During the second round both i_1 and i_2 become senders, and two other machines i_3 and i_4 receive the information. Information can therefore be broadcast to all machines in $\lceil \log n \rceil$ rounds. Broadcasting by data dissemination requires at least $\lceil \log n \rceil$ rounds. Simultaneous broadcasts can piggyback data onto the same messages.

The situation becomes a more complicated when the following conditions must be met.

(a) No matter which machine starts broadcasting, the information should reach all n machines in $\lceil \log n \rceil$ rounds.

(b) No matter at what round a machine starts broadcasting, the information should reach all n machines in $\lceil \log n \rceil$ rounds.

(c) The scheme should be able to detect and/or tolerate certain faults.

(d) The system should be able to dynamically reorganize after certain faults.

The scheme proposed by Alon *et al.* [1] meets conditions (a) and (b) when n is a power of 2 or one of a family of primes. For general n, they adopt a quotient emulation [6] that requires $2\lceil \log n \rceil$ rounds. Their scheme can detect any single-machine failure in n rounds. (A failed machine no longer sends messages.) They propose analytic solutions for faulty systems as an area worthy of further research.

In this paper we present an optimal scheme for disseminating information. Our scheme meets conditions (a) and (b) for any n. In addition, our scheme tolerates single-machine failure if the number of rounds used for broadcasting information is $\lceil \log n \rceil + 2$. Our scheme allows the system to be reorganized in $\lceil \log n \rceil + 2$ rounds to excise a failed machine.

The underlying model that we will use assumes that any permutation of messages can be realized in one round. We consider such a permutation of messages a basic communication event. The token ring network and the ethernet are particularly suitable as implementations for our model.

However, a restricted network with $\lceil \log n \rceil$ connecting links for each machine is sufficient for our data-dissemination procedure; after each $\lceil \log n \rceil$ rounds, our scheme repeats the destination pattern.

The rest of the paper is organized as follows. Section 2 presents our scheme for data dissemination. Section 3 shows how our scheme can be modified to tolerate single faults. Section 4 discusses reorganizing the system after failure. We summarize these results and draw some conclusions in Section 5.

Broadcast

Our scheme for disseminating information is for each machine i, $0 \le i < n$, to execute the following procedure.

```
procedure Disseminate;
loop
  -- each iteration is one run
  for round := 0 to ⌈log n⌉-1 do
    -- each iteration is one round
    message := new+old broadcast data
    send message to machine (i+2^round) mod n
  end -- for
end -- loop
```

The information sent by machine i during any time t is a combination of

- New information that i has decided to broadcast starting at time t.
- Information received by i during the previous $\lceil \log n \rceil -1$ rounds that must still be transmitted to other machines.

We will not be concerned with how messages are packaged and how i maintains its data structures to show which information needs to be sent on any round. We will assume that messages are long enough to hold all the information that must be sent. If a basic unit of information uses b bytes, and each machine may only originate one broadcast per round, then the longest message needed will be $2bn$ bytes. However, many applications can reduce this length by combining information from several simultaneous broadcasts. To produce an average, for example, a broadcast message need only include the sum of the values provided by all the broadcast sources.

We will call a set of $\lceil \log n \rceil$ rounds a **run**. The following chart shows the destinations for the 3 rounds that constitute a run when $n = 6$.

round	source					
	0	1	2	3	4	5
0	1	2	3	4	5	0
1	2	3	4	5	0	1
2	4	5	0	1	2	3

If machine 2 wishes to disseminate some information at round 0, it will send it to machine 3. At round 1, machines {2, 3} will send the information to {4, 5}. At round 2, machines {2, 3, 4, 5} will send the information to {0, 1, 2, 3}, at which point all 6 machines will have the desired information. In fact, machines {2, 3} will have seen it twice. If this is a problem (and it would be for computing global statistics such as averages), the messages can include a unique broadcast serial number (in this case, picked by machine 2) that can be checked for duplicates.

All machines must agree on the current round within a run. Either they can refer to a global clock, or they can agree that each machine will send exactly one message per round even if no broadcasts are current.

Theorem 1: Procedure `Disseminate` will broadcast information from any source to all destinations within any consecutive $\lceil \log n \rceil$ rounds.

Proof: Any broadcast started by machine i arrives at machine j in round s if and only if a broadcast started by machine 0 arrives at machine $(j-i) \bmod n$ in round s, so without loss of generality, we may assume the source is machine 0. After $\lceil \log n \rceil$ rounds, the broadcast has reached machines $(0[+2^1]\,[+2^2]\ \ldots\ [+2^{\lceil \log n \rceil -1}]) \bmod n$, where square brackets indicate that the term is optional. The order in which these machines are reached depends on which round in a run is current when the broadcast starts, but the set of reached machines is the same in any case. Clearly, this set includes all machines $0 \le i < n$. □

In procedure `Disseminate`, we can view all machines as arranged in a cycle, as shown in Figure 1. The (asymmetric) distance from machine p to machine q is $(q-p) \bmod n$. If broadcast starts at round 0, then after round i, machines within distance 2^i from p will receive information from p. The pattern is not so clear for broadcasts that start at different rounds. Alon *et al.* [1] realized that `Disseminate` will finish broadcasting if it is started at round 0, whereas Theorem 1 proves a stronger fact.

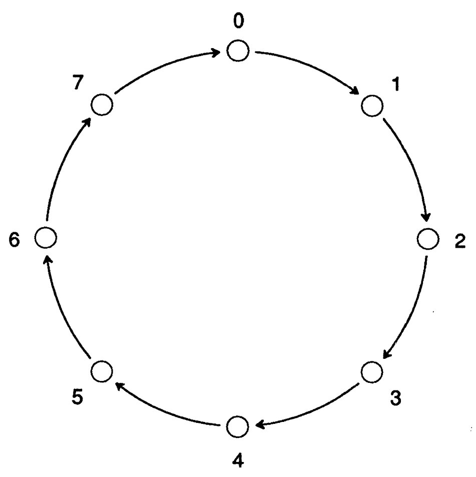


Figure 1: Conceptual arrangement of machines

Fault tolerance

We assume that machine failures prevent messages from being sent from the failed machine, but that no other indication of failure is given. This failure model is intermediate between immediately globally visible failures, which are trivial to deal with, and Byzantine failures [7] , which are quite difficult.

The disseminating process of `Disseminate` assumes a tree shape, as shown in Figure 2. We will call this tree the **broadcasting tree** of machine 0 with depth $\lceil \log n \rceil$ starting at round 0.

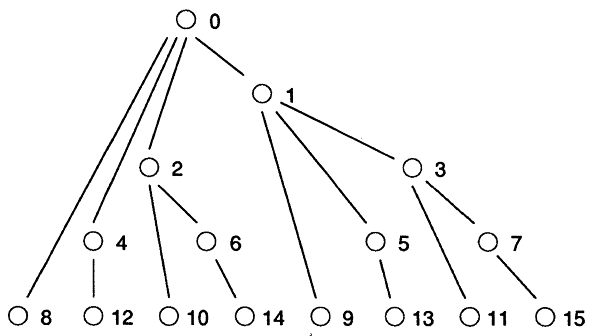


Figure 2: Broadcasting tree of machine 0 with 4 rounds starting at round 0.

A failed machine will cast a shadow in this tree, preventing its descendants from getting information originating from the root of the broadcasting tree. In order to transfer information from the root to all nodes, we have to use more than $\lceil \log n \rceil$ rounds. Extra rounds can circumvent the failed machine. In the following theorem we show that if exactly one machine fails, a message from any other machine can reach all working machines within any consecutive $\lceil \log n \rceil + 2$ rounds of `Disseminate`.

Theorem 2: Procedure `Disseminate` will broadcast information from any source to all destinations within any consecutive $\lceil \log n \rceil + 2$ rounds if exactly one machine has failed.

Proof: Without loss of generality, machine 0 is the source of the broadcast. Suppose machine 0 starts broadcast at round i, and the information first reaches the failed machine f at round j, where $0 \le i, j < \lceil \log n \rceil$. Consider the following cases:

Case 1: $0 \le i \le j < \lceil \log n \rceil$. After rounds $i, i+1, \ldots j, \ldots, \lceil \log n \rceil - 1, 0, 1, \ldots, i-1$, machines that have not yet received information from machine 0 are in $S = \{f \le m \mid m = f\,[+2^{j+1}] \ldots [+2^{\lceil \log n \rceil - 1}][+2^0] \ldots [+2^{i-1}]\}$. No pair of machines in S is separated by exactly 2^i. In the next round, round i, every machine in S will receive information from a source 2^i places earlier in the cycle, and that source is not in S. Therefore, that source already has the information derived from machine 0. In this case, $\lceil \log n \rceil + 1$ rounds suffice for broadcasting information.

Case 2: $0 \le j < i < \lceil \log n \rceil$. After rounds $i, i+1, \lceil \log n \rceil - 1, 0, 1, \ldots, j, \ldots, i-1$, the machines that have not yet received information from machine 0 are in $S = \{m \mid m = f\,[+2^{j+1}] \ldots [+2^{i-1}]\}$. When $i < \lceil \log n \rceil - 1$, no two machines in S are separated by exactly 2^i, so broadcasting can be accomplished with one more round. When $i = \lceil \log n \rceil - 1$, it is possible that two machines in S are separated by exactly 2^i (see the example following this proof). In this situation, two more rounds will finish the broadcast. The reason is that when $i = \lceil \log n \rceil - 1$, the second broadcast round is round 0. A broadcast that starts at round 0 would require $\lceil \log n \rceil + 1$ rounds to complete in the face of a single failure, as shown in Case 1. □

Here is an example that demonstrates the necessity of $\lceil \log n \rceil +2$ rounds. Let $n=10$, $i=3$ (the round at which broadcast starts), $f=1$ (the failed machine). Broadcast is initiated by machine 0. The following chart shows that 6 rounds are required for broadcast to complete. A mark (●) indicates that the given machine has received the broadcast.

round	machine									
	0	1	2	3	4	5	6	7	8	9
3	●	f							●	
0	●	f							●	●
1	●	f	●						●	●
2	●	f	●	●	●		●		●	●
3	●	f	●	●	●		●	●	●	●
0	●	f	●	●	●	●	●	●	●	●

The analysis presented in this section shows that by using two more rounds for broadcasting our scheme can tolerate single faults.

Reorganization

After a machine f fails, it can be advantageous to reorganize and degrade an n-machine system to an $(n-1)$-machine system. Reorganization can be accomplished as follows. If $f=n-1$, the rest of the machines simply assume that there are only $n-1$ machines left. If $f \neq n-1$, machine $n-1$ will assume the identity of f. Depending on the communication medium used, destination tables might need to be modified on every machine. (We require that failures never partition the network.)

It can happen that $n=2^j+1$ for some j, in which case the reduction in machines leads to a new value for $\lceil \log n \rceil$. The current round number may be invalid for this new value. Therefore, we will reset the round to 0 after reorganization.

We must resolve two technical details to accomplish reorganization. The first is achieving simultaneous reorganization, and the second is protecting active broadcasts during reorganization.

Simultaneous reorganization

Reorganization must occur simultaneously on all machines or chaos will result. The following method achieves this aim. If a single machine f should fail, one machine, say i, will notice that fact during the next round as an expected message fails to arrive. A different machine, say j, will notice the failure during the subsequent round as its expected message fails to arrive. Meanwhile, i can report the failure to a third machine k during the same period. Although both i and j notice that f has failed, only i gets the correct failure time. The failure time of f noticed by j is later than that noticed by i. During later rounds, if machine m receives word of the failure from both i and j, m can discard the notice from j because the failure time reported by i is earlier than that from j. By Theorem 2, a failure notice from i will be received by every functioning machine in $\lceil \log n \rceil +2$ rounds. Every such machine can reorganize its tables effective $\lceil \log n \rceil +3$ rounds after the failure occurred. This scheme is expressed by the following procedure.

```
procedure Reorganize;
-- reorganization during broadcast
  var
    i : integer := whatever;
      -- identifier of this machine
    n : integer := whatever;
      -- number of machines
    clock : integer := 0;
      -- monotone increasing every round
    f : integer; -- which machine failed
    failtime : integer := ∞;
      -- when it failed: based on clock
    reorganize : integer := -1;
      -- time reorganization scheduled
  loop -- each iteration is one run
    for round := 0 to $\lceil \log n \rceil -1$ do
      if reorganize = clock then
        -- time to reorganize
        if i = n-1 then
          i := f;
        endif;
        failtime := ∞;
        n := n-1;
      endif;
      message := traffic, f, failtime;
      send message to $(i+2^{round}) \bmod n$
      accept message from $(i-2^{round}) \bmod n$
      if no message arrives then
        -- a failure has occurred
        newfail := $(i-2^{round}) \bmod n$
        newfailtime := clock - 1;
      else
        traffic, newfail, newfailtime :=
          message;
      endif;
    if newfailtime < failtime then
      -- discard old value
      failtime := newfailtime;
      f := newfail;
    endif;
    if failtime < ∞ then
      reorganize := failtime +
        ⌈log n⌉ + 3;
    endif;
    clock := clock + 1;
  end -- for loop; one round
end -- loop; one run
```

Based upon the above analysis we have:

Theorem 3: The system can be reorganized in $\lceil \log n \rceil + 3$ rounds after a single failure.

Proof: If machine f fails at round i, then at round $i+1$, some machine notices the failure and the failure time. By Theorem 2, this information will be known to all machines in an additional $\lceil \log n \rceil + 2$ rounds. Therefore at the end of round $\lceil \log n \rceil + 3$, every machine is ready for reorganization. □

The first machine p that observes the failure starts a failure broadcast. The relation between this p and f enables us to avoid the ($\lceil \log n \rceil + 2$)-round case (Case 2 of the proof of Theorem 2). We can prove the following stronger result.

Theorem 4: The system can be reorganized in $\lceil \log n \rceil + 2$ rounds after a single failure.

Proof: The only situation where $\lceil \log n \rceil + 2$ rounds are required for broadcasting with a single failure occurs when a machine starts broadcast at round $\lceil \log n \rceil - 1$ (see the proof of Theorem 2). This happens when f fails at round $\lceil \log n \rceil - 3$, machine p discovers the failure at round $\lceil \log n \rceil - 2$, and starts broadcast at round $\lceil \log n \rceil - 1$.

In this case the distance from f to p is $2^{\lceil \log n \rceil - 2}$ and the distance from p to f is greater than $2^{\lceil \log n \rceil - 2}$ (recall that the distance is asymmetric). Ignore the first round in broadcast, that is, round $\lceil \log n \rceil - 1$. Machine f is a leaf in the broadcasting tree of machine p starting at round 0 and therefore will not cast shade on any machine in this tree. Thus in $\lceil \log n \rceil + 2$ rounds after failure every machine has the correct failure information□

Active broadcasts

The other detail to be worked out is the fate of broadcasts active during failure and reorganization. Reorganization is likely to introduce chaos into any broadcasts that are active. For this reason, machines should not start new broadcasts if they know that a reorganization is scheduled within the next $\lceil \log n \rceil + 2$ time units. (If a reorganization is scheduled, there must be a failed machine, in which case broadcast can take that long to finish.) Any broadcast that was inadvertently started within this window should be restarted.

A failure that occurs at time t could damage broadcasts started as early as time $t - \lceil \log n \rceil + 1$. Two strategies can cope with these damaged broadcasts. The first strategy restarts such broadcasts after the error has been repaired. Broadcasts may therefore require as long as $3\lceil \log n \rceil$ time to complete. The first $\lceil \log n \rceil$ is for the original broadcast that fails near the end because of a broken machine. The second $\lceil \log n \rceil$ is needed to inform the world about the failure, and the third $\lceil \log n \rceil$ is to run the broadcast again.

The second strategy uses $\lceil \log n \rceil + 2$ rounds for each broadcast to tolerate single faults so that broadcast need not be restarted when a machine fails. The first strategy is more efficient on the average if failures are rare, but can suffer long delays when failures occur. The second strategy uses $\lceil \log n \rceil + 2$ rounds for every broadcast, but it is a useful strategy for real-time systems.

Conclusions

We have presented an optimal data-dissemination algorithm for machines capable of sending messages to each other. It improves previous results in that it does not restrict the number of machines and accommodates single-machine failure and reorganization.

Although our data-dissemination algorithms are described for point-to-point networks, they are also suited to multi-machine organizations with broadcast media such as token-ring network and ethernets [8]. In these networks, each message can potentially reach all machines, although usually only the intended destination machine actually reads any message. Ordinary broadcast by n processes requires n messages, but each machine must receive $n-1$ messages, leading to $O(n^2)$ total work. In comparison, dissemination requires only $O(n \log n)$ total work. The reduction is due to piggybacking messages. In some applications, such as finding average load, piggybacking does not require increasing the message size.

Several open questions remain. Multiple failures could prevent broadcasts from reaching from some sources to some destinations altogether. We know that one failure never has this property. What is the minimum number of failures that does?

Our reorganization scheme is synchronous. Is there an asynchronous reorganization scheme that preserves broadcasts that are underway?

Acknowledgements

We would like to thank Debra Hensgen for helpful discussions.

References

1. N. Alon, A. Barak, and U. Manber, "On disseminating information reliably without broadcasting," *Proceedings of the 7th International Conference on Distributed Computing Systems*, pp. 74-81 (September 1987).
2. Z. Drezner and A. Barak, "A probabilistic algorithm for scattering information in a multiprocessor system," Technical report CRL-TR-15-84, University of Michigan Computing Research Laboratory (March 1984).
3. C.E. Leiserson and J.B. Saxe, "Optimal synchronous systems," *Proceedings of the 22nd Annual IEEE Symposium on Foundations of Computer Science*, pp. 23-36 (1981).
4. M. Livny and U. Manber, "Distributed computation via active messages," *IEEE Trans. on Comput.* **C-34**(12) pp. 1185-1190 (December 1985).
5. D. Nassimi and S. Sahni, "Data broadcasting in SIMD computers," *IEEE Trans. Comput.* **C-27**(2) pp. 2-7 (1979).
6. J.A. Fishburn and R.A. Finkel, "Quotient networks," *IEEE Trans. on Comput.* **C-31**(4) pp. 288-295 (April 1982).
7. R. Cristian, H. Aghili, and R. Strong, "Atomic broadcast from simple message diffusion to byzantine agreement," Technical report RJ 4540, IBM San Jose Research Center (December 1984).
8. A.S. Tanenbaum, *Computer networks,* Prentice-Hall (1981).

Multicast Routing in Spanning Bus Hypercubes

Philip McKinley

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

ABSTRACT

This paper investigates communication among groups of cooperating processes in 3-dimensional spanning bus hypercubes. Requiring processes to send a separate copy of each message to every group member may result in high bandwidth consumption and network traffic congestion. Multicast virtual circuits provide an alternative mechanism for process group communication. Unfortunately, constructing a single, optimal multicast tree for a group of nodes in a 3-dimensional spanning bus hypercube is an *NP*-hard problem. In this paper, several heuristic algorithms for this problem are evaluated and compared. We introduce a greedy algorithm with complexity linear in the group size. This algorithm performs better than other algorithms with respect to both tree cardinality and average message propagation delay.

1. Introduction

A multicomputer is a collection of computers designed to operate as parallel processors [1,2]. In a multicomputer, the individual processors communicate by passing messages via a communication network. A D-dimensional, W-wide spanning bus hypercube [3] is a multicomputer architecture in which each processor, or *node*, is connected by ports to D buses, each bus spanning a different dimension in the hypercube address space. Each bus is connected to exactly W nodes. A 3-dimensional, 3-wide spanning bus hypercube is shown in Figure 1.

Until recently, the primary disadvantage of the spanning bus hypercube architecture has been that the width, W, is limited by the bandwidth of the buses. However, continuing advances in communication technology, particularly optical fibers, have largely obviated this problem. In particular, star couplers [4] offer an efficient way to organize optical fibers as multiple-access media, or *buses*, that may be connected together to form a spanning bus hypercube network. Figure 2 gives a representation for a 2-dimensional, 3-wide spanning bus hypercube constructed from optical stars. Networks constructed from multiple optical stars have been studied [5–7] in the context of local area networks, and we [8] have investigated ways to exploit their unique properties in distributed system design. Marhic [4] suggested their use in interconnection networks for parallel processors.

The small diameter and average internode distance, combined with low cost, make the spanning bus hypercube architecture attractive for parallel processing [9]. The topology is particularly well-suited to *multicast* communication. In contrast to *unicast* transmission, which involves a single destination node, *multicast* allows for one source node to communicate with more than one destination node. *Broadcast* is a special case of multicast in which the data is delivered to all nodes in the network. Parallel applications such as particle dynamics calculations and image processing often require communication among groups of processes [10]. Hence, efficient multicast is important.

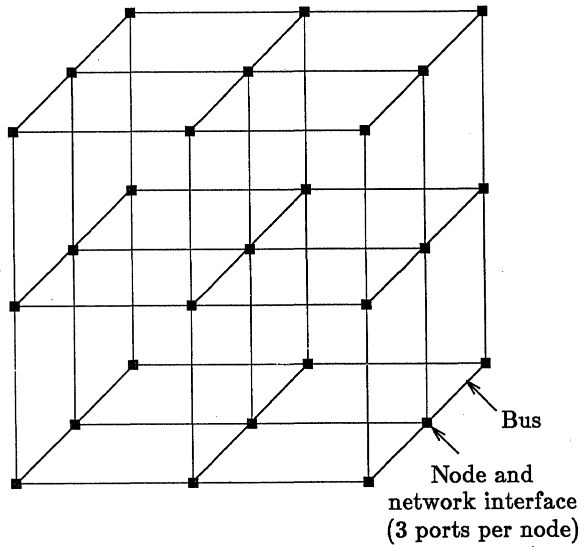


Figure 1 3-d, 3-w spanning bus hypercube

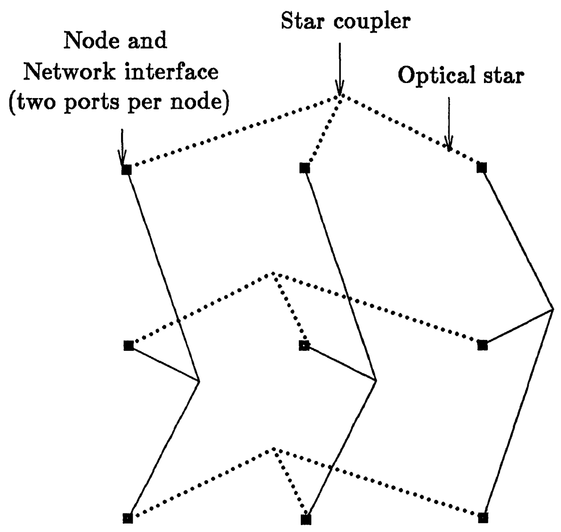


Figure 2 2–d, 3–w spanning bus hypercube

In this work we address the issue of multicast routing in a 3–dimensional spanning bus hypercube. Three dimensions allow a large number of nodes to be connected in a network with a very small diameter. We assume the system to be a general purpose machine capable of executing multiple applications simultaneously, with multiprogramming at processor nodes. Our primary concern is with communication among process groups whose members are scattered throughout the network. Such a situation may arise for two reasons. First, the nature of a particular application may require communication among subgroups of processes whose memberships change dynamically. Second, the assignment of process group members to processors upon initiation of a parallel application depends upon the assignments of the applications already running and their current activities. Because these conditions are dynamic, the processors most eligible to execute the new application may be widely dispersed in the network. We assume that communication among process group members is frequent relative to the lifetime of process groups. In addition, any group member may send messages to the other group members, that is, group communication is symmetric.

This work does not concern management issues for process groups, such as dynamic group membership, or the details of message buffering and flow control. These functions would be provided by a process group communication mechanism such as *multicast channels* [10]. Rather, the routing algorithms described in this paper can be considered as a method for efficiently supporting such a mechanism in a spanning bus hypercube architecture.

The simplest method of multicast is to require that a process send a separate copy of each message to every other process group member. Such a strategy may result in high bandwidth consumption and traffic congestion. An alternative solution for multicasting is to support *tree forwarding*, in which virtual circuits are established along the branches of a tree that spans the group members. Data travels from the source node along the branches of the tree. Once a multicast tree has been constructed, forwarding and duplication of data is accomplished using routing tables at network nodes.

To reduce bandwidth consumption and traffic congestion, it is desirable to minimize the number of branches (communication links) in the tree. This problem is *NP*–hard for arbitrary point–to–point networks, but Wall [11] has investigated heuristic algorithms that perform well in practice. The multicast tree problem is also *NP*–hard for spanning bus hypercubes [12] and hence we investigate heuristic algorithms. As in other work concerning multicast trees in bus–based networks [13,14], we are particularly interested in algorithms that exploit the broadcast nature of the media. In Section 2, we give some background on the problem needed in our subsequent discussion by briefly describing several approaches to multicast routing. Section 3 describes and compares three algorithms for the construction of multicast trees. Finally, in Section 4, we present conclusions and summarize our work.

2. Multicast Routing Schemes

The multicast routing scheme used in a multicomputer directly affects the performance of process group communication and, hence, the performance of parallel applications. Several multicast routing schemes have been proposed. An extensive discussion of these strategies can be found in [15].

Separate Addressing

Separate addressing is the simplest multicast technique. In this scheme, a separate copy of the message is sent to each destination. This strategy can be implemented directly atop a unicast transmission protocol. In hypercube topologies, unicast routing is straightforward [3]. Each node address consists of D coordinates, one for each dimension of the hypercube. For example, (x_2,x_1,x_0) is an address for a node in a 3–dimensional spanning bus hypercube. Nodes in the hypercube shown in Figure 1 are assigned addresses (0,0,0) through (2,2,2), as shown in Figure 3. We shall refer to a bus by an address with a missing coordinate to indicate the direction of the bus. For example, bus (0,–,2) lies in the x_1 direction and is in the 0th x_2 plane and the 2nd x_0 plane.

When a message is being forwarded from one node to another, differing coordinates between the present address and the destination address determine the next bus on which to forward the message. For example, consider the spanning bus hypercube shown in Figure 3. If coordinates are resolved in the order x_0, x_1, x_2, then when node (0,0,0) sends a message to node (1,2,1), the message will first traverse bus (0,0,–), then bus (0,–,1), and finally bus (–,2,1). The order in which coordinates are resolved may be varied, and does not affect the number of buses that must be traversed to reach the destination.

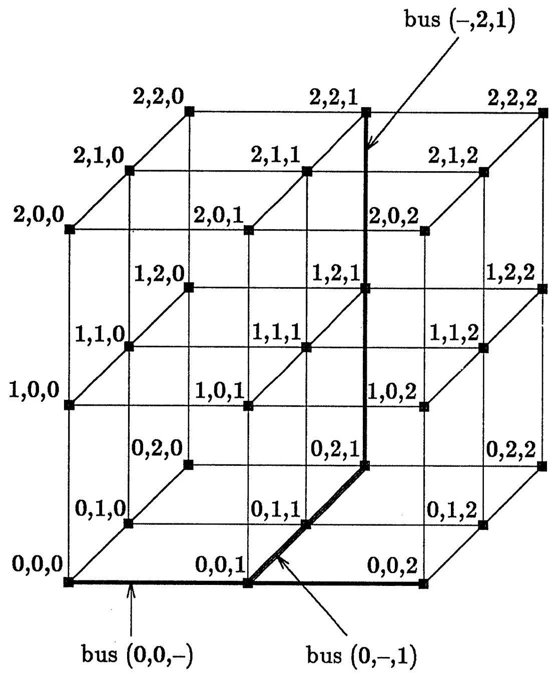


Figure 3 3–d, 3–w spanning bus hypercube

The primary disadvantage of separate addressing as a multicast technique is that multiple copies of a message must be sent and may actually traverse the same communication links. This results in excessive bandwidth consumption and congestion at intermediate nodes. If we call the transmission of a packet on any bus a *bus visit*, then for separate addressing the number of bus visits is proportional to the group size. Given a source node and g destination nodes distributed uniformly in a 3–dimensional, w–wide spanning bus hypercube, the average number of bus visits for a multicast using separate addressing can be computed directly. We partition the potential destination nodes into three groups according to distance from the source node along a shortest path: $3(w-1)$ nodes are one hop away, $3(w-1)^2$ nodes are two hops away, and $(w-1)^3$ nodes are three hops away. There are w^3-1 potential destination nodes, and hence the average number of bus visits required to send to g destinations is:

$$\frac{3(w-1)\cdot 1 + 3(w-1)^2\cdot 2 + (w-1)^3\cdot 3}{w^3-1}\cdot g = \frac{3w^2 g}{w^2+w+1}.$$

Despite its drawbacks, separate addressing is the best multicast routing strategy for small groups, especially if the rate of group message traffic is low [16]. Any overhead accompanying a more complex scheme is not justified.

Multidestination Addressing

One popular multicast scheme that avoids sending multiple copies of a message along the same communication links is *multidestination addressing* [17–20], in which a small number of multiply–addressed packets are sent for each multicast. When a packet arrives at an intermediate node, its destination addresses are apportioned among multiple copies such that destinations along the same route share the same copy.

For a 3–dimensional, w–wide spanning bus hypercube, the number of bus visits per multicast for multidestination addressing can also be directly computed. We assume, as before, that member nodes are uniformly distributed and that packets traverse shortest paths along links in x_0, x_1, x_2 order. Figure 4 depicts with bold lines the buses that may be traversed by multicast messages sent from node *s*. The buses shown with dotted lines would not be used. The probabilities of traversing buses in each direction are, respectively:

$$P_{x_0}(w,g) = 1 - \binom{w^2-1}{g}\binom{w^3-1}{g}^{-1}$$

$$P_{x_1}(w,g) = 1 - \binom{w^3-w^2+w-1}{g}\binom{w^3-1}{g}^{-1}$$

$$P_{x_2}(w,g) = 1 - \binom{w^3-w}{g}\binom{w^3-1}{g}^{-1}$$

Since the numbers of potential x_0–, x_1–, and x_2–buses to be visited are 1, w, and w^2, respectively, the average number of bus visits per multicast is:

$$P_{x_0}(w,g) + w\cdot P_{x_1}(w,g) + w^2\cdot P_{x_2}(w,g) =$$

$$w^2+w+1 - \frac{w^2\binom{w^3-w}{g} + w\binom{w^3-w^2+w-1}{g} + \binom{w^2-1}{g}}{\binom{w^3-1}{g}}.$$

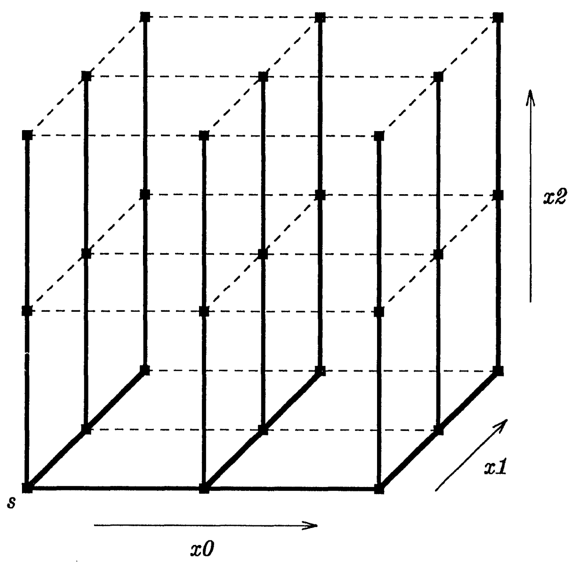


Figure 4 Routing with multidestination addressing

Figure 5 compares the average number of bus visits for separate addressing and multidestination addressing in a 3–dimensional, 8–wide spanning bus hypercube for various group sizes. While the number of bus visits in the separate addressing case increases linearly with the group size, the number of buses in multidestination routing asymptotically approaches w^2+w+1 or, in this case, 73.

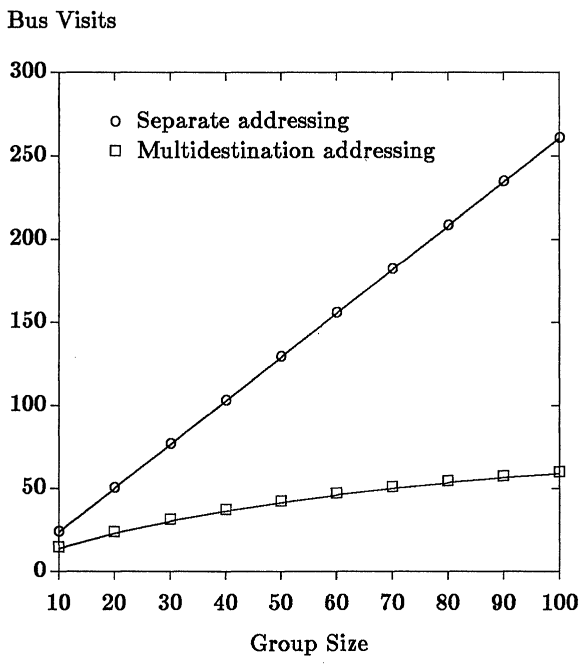


Figure 5 Bus visit comparison
3–d, 8–w spanning bus hypercube

Variable length packet headers, required to contain multiple destination addresses, and computation time for apportioning these addresses are the major drawbacks of multidestination addressing [15]. Both of these shortcomings become more serious as group size increases.

Tree Forwarding

An alternative approach is *source routing*, in which the route is placed in the packet header. Although normally intended for larger networks [21,22], this strategy has advantages for multicast routing in a regular topology, such as a spanning bus hypercube, where a relatively complex multicast path can be efficiently encoded. However, a serious drawback is the fact that, without state information at intermediate nodes, flow control information and acknowledgements must be sent individually from each destination to the source, potentially causing congestion at the source and at intermediate nodes. In addition, the scheme must be robust enough to handle inoperative network components, particularly switches at intermediate nodes.

Tree–forwarding avoids these problems by constructing multicast virtual circuits between the processors on which process group members reside. A group identifier, rather than a list of destinations, in the packet header can be used by intermediate nodes to forward the packet, based on state information maintained along the route. This approach provides more efficient end–to–end control information by merging flow control and acknowledgement messages as they proceed from the destinations to the host [10].

The number of buses, or cardinality, of a multicast tree rooted at a particular node matches the number of bus visits required for a multicast from that node using multidestination addressing. *Single*–tree forwarding attempts to minimize bandwidth consumption, state information and maintenance costs by constructing a single tree that is used by all the nodes in a multicast group. In *multiple*–tree forwarding, each member uses a separate tree to multicast to the other members. Each tree is constructed by merging the shortest paths from the individual source node to the other nodes. The length of a path may be measured in number of hops or expected delay.

The major problem with multiple–tree forwarding is that the size of the state information required is proportional to the square of the group size [15]. Also, this scheme is not amenable to dynamic group membership, as ***each*** tree must be modified when a node joins or leaves the group. Although the average point–to–point delay is usually greater for single–tree forwarding than for multiple–tree forwarding, this difference may not be significant in a highly connected network such as a spanning bus hypercube. More importantly, this shortcoming is outweighed by simpler tree construction, tree maintenance, and exchange of control information. Hence, we are concerned only with single–tree forwarding algorithms.

3. Single–Tree Routing Schemes

When single–tree forwarding is used to multicast data, the routing problem reduces to that of constructing *good* multicast trees. Good trees have small cardinalities and reasonable delays between each pair of nodes. In this section we evaluate and compare algorithms for constructing good multicast trees in 3–dimensional spanning bus hypercubes. As pointed out earlier, we are interested in algorithms that exploit the broadcast nature of the media. The first two algorithms are based on the shortest paths between pairs of member nodes. The last algorithm, however, accounts for the fact that multiple group members may reside on the same bus.

Simple Shortest Path Routing

One way to multicast from a source to a group of processes is to merge the point–to–point paths into a single tree and construct virtual circuits along its branches. We call this *simple* shortest path routing. A different tree can be constructed for each member of a process group. For single–tree forwarding, one of these trees is (arbitrarily) chosen for communication within the group. The trees are not identical with regard to their cardinality or the number of hops between nodes, so it is possible that the best tree will not be chosen. Considering all possible roots and then choosing the best one requires time quadratic in the group size.

An alternative approach is to arbitrarily choose a root but consider alternate orders in which to resolve address components. For example, in a 3–dimensional spanning bus hypercube there are six possible orders in which to resolve references, corresponding to the six permutations of x_0, x_1 and x_2. For single–tree forwarding, all of these computations are carried out and the single best tree is chosen for communication between the group members. We call this scheme *ordered* shortest path routing.

Figure 6 compares the cardinality (that is, bus visits per multicast) of multicast trees for simple shortest path routing and ordered shortest path routing over a range of group sizes in a 3–dimensional, 8–wide spanning bus hypercube. Although difficult to see because they are so small, 99 percent confidence intervals are plotted in this and all other graphs. Ordered shortest path routing provides an improvement over simple shortest path routing.

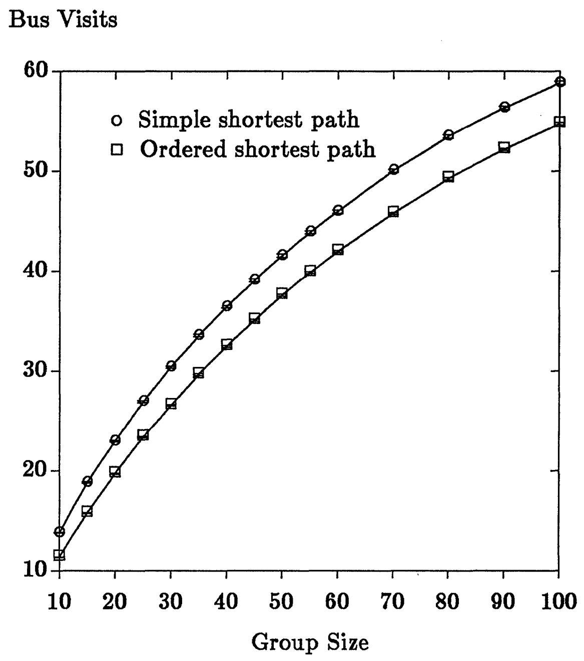


Figure 6 Bus visit comparison
3–d, 8–w spanning bus hypercube

Centered Routing

An alternative to arbitrarily selecting one of the member nodes to be the root of the tree is to select a node that is centrally located within the group. We call this scheme *centered* shortest path routing. This technique is similar to the technique known as *center–based forwarding*, studied for point–to–point networks [11]. Selecting the center of the group may be done in several ways. The mechanism we have used is simple and requires time linear in the group size. First, the algorithm identifies the three planes, one in each dimension, that contain the most group members. The node at the intersection of these planes is chosen as the root of the multicast tree. This node need not be a member of the group, but may only be called upon to forward packets. Next, by resolving addresses in different orders, the six possible trees are computed. Finally, these trees are compared and the tree with the smallest number of buses is chosen.

Figures 7 and 8 compare the cardinality and average delay of ordered routing and centered routing. While centered routing requires slightly more buses on average, the average number of hops between nodes is fewer than for ordered shortest path routing.

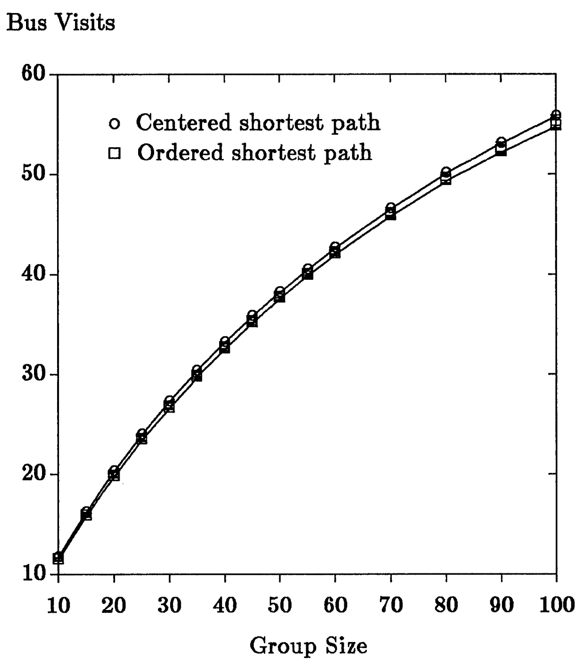


Figure 7 Bus visit comparison
3–d, 8–w spanning bus hypercube

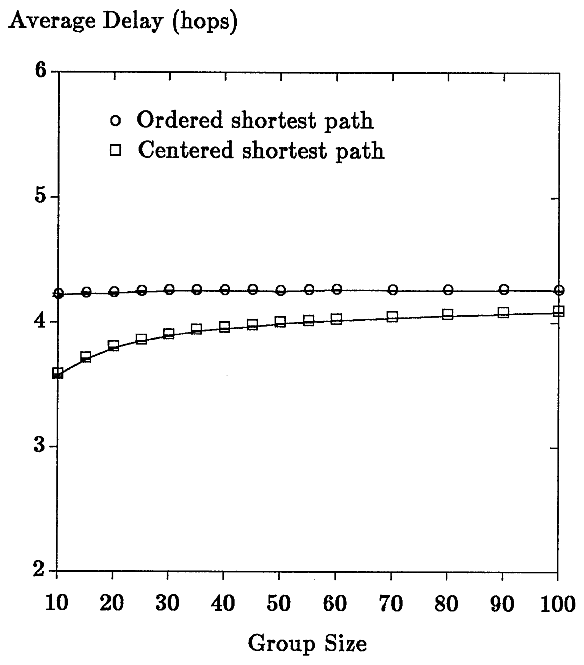


Figure 8 Average delay comparison
3–d, 8–w spanning bus hypercube

Greedy Algorithm

In the preceding discussion, we have seen that multicast tree construction based on shortest paths between pairs of nodes is sensitive to which node is the root of the tree, as well as to the order in which address components are resolved in computing the shortest paths. In this section, we describe a relatively simple *greedy* algorithm that does not require construction of multiple trees and that performs better than those algorithms previously described. In the greedy algorithm, the basic metric used to construct multicast trees is the number of member nodes on each bus. Buses connected to many member nodes are more likely to be included in the multicast tree. The intuitive appeal of such an approach is that, unlike algorithms based on the distance between pairs of member nodes, greedy algorithms are better able to exploit the fact that the media are buses.

In an earlier paper [13], we have shown that the performance of a greedy algorithm is very close to optimal for 2–dimensional grids of buses, that is, 2–dimensional spanning bus hypercubes. Since each plane of a 3–dimensional spanning bus hypercube is a 2–dimensional spanning bus hypercube, the algorithm initially identifies the three orthogonal planes that contain the most group members. We call these the *primary* planes. The group members in the three primary planes are covered using the 2–dimensional spanning bus hypercube multicast algorithm. Next, the group members not in primary planes are covered. If the bus used to cover such a node does not intersect one of the primary planes at a node that is already covered, an additional bus in the primary plane is included in the multicast tree to guarantee connectedness. The algorithm is given in Figure 9.

Figures 10 and 11 compare the cardinality and average delay performance for simple shortest path routing, centered routing, and the greedy algorithm. All have complexity linear in the group size. The greedy algorithm provides the best cardinality performance and the best average delay performance for large groups. It is interesting that the delay for this algorithm, after initially increasing, decreases as group size increases. This results because the tree has no designated root. Point–to–point distances continue to decrease, while in the other schemes packets must more often go through the root node. When routing is based on shortest paths from a particular node, a shorter path between two group members is less likely to be included in the tree.

Greedy algorithm for multicast tree construction
Input: A 3–d sbh S and a subset G of its nodes.
Output: A multicast tree T for G.

```
begin
   find primary planes p0, p1 and p2
   let inP = { g ∈ G | g resides in p0, p1 or p2 }
   let outP = G - inP

   /* cover members in the primary planes first */
   for each of p0, p1 and p2 do
   begin
      /* cover nodes with the 2-d greedy algorithm */
      for each g ∈ inP that is in the plane do
      begin
         select bus b of g with most uncovered members
         T = T ∪ {b}
         inP = inP - { r ∈ inP | r lies on b }
      end
   end
   if p0, p1, and p2 are not connected then
      connect them using buses in their intersections
   endif

   /* handle members not in the primary planes */
   for each g ∈ outP do
   begin
      select bus b of g with most uncovered members
      T = T ∪ {b}

      /* connect b to a primary plane */
      let b intersect one of p0, p1 or p2 at node c
      if c is not covered then
         let t be a bus of c in the primary plane
         T = T ∪ {t}
      endif
       outP = outP - { r ∈ outP | r lies on b }
   end
end
```

Figure 9 Greedy Algorithm

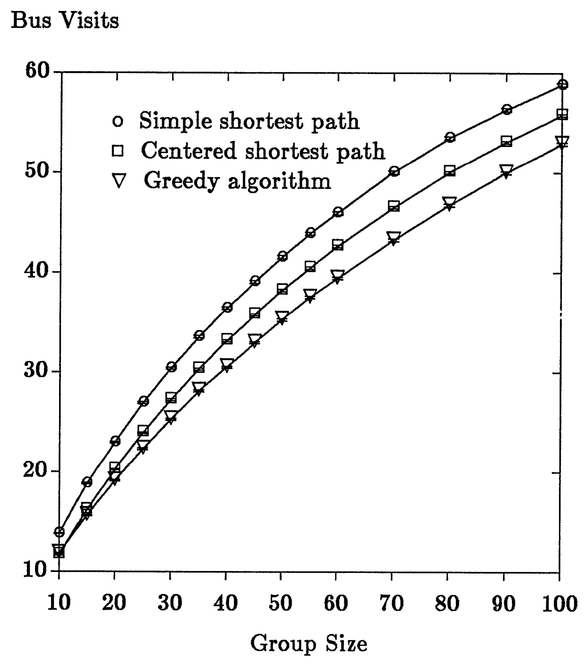


Figure 10 Bus visit comparison
3–d, 8–w spanning bus hypercube

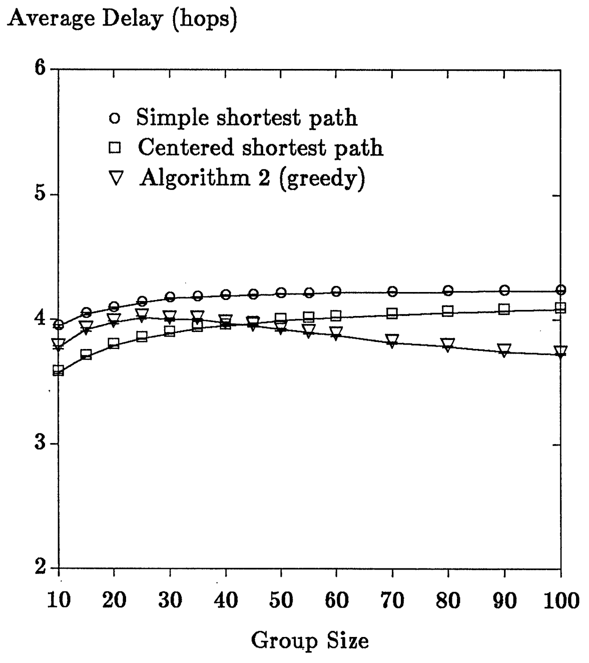


Figure 11 Average delay comparison
3–d, 8–w spanning bus hypercube

4. Summary

Bus-based multicomputer networks facilitate multicast communication, an important facet of many parallel algorithms. With recent advances in optical fiber technology, bus-based interconnection networks are receiving new attention. Separate addressing is inefficient unless the process group is small, and multidestination addressing is difficult to implement and may cause unreasonable delays in packet forwarding. Multicast virtual circuits offer a solution to these problems. While multiple-tree forwarding is not attractive because of the large table space required, single-tree forwarding requires space linear in the group size. Although average delay between group members is usually larger when using a single tree than when using a separate tree for each group member, the average delay can still be reasonably small in a bus-based network with high connectivity, such as a spanning bus hypercube.

In this paper we have compared multicast tree construction algorithms based on greedy methods with those that do not account for the broadcast nature of the media. We have shown that the former performs favorably while maintaining linear complexity. We are currently evaluating other multicast routing algorithms for spanning bus hypercubes and other bus-based networks.

Acknowledgements

This research was conducted at the University of Illinois Advanced Construction Technology Center, funded by the United States Army Research Office under the Department of Defense-University Research Initiative Program.

References

1. D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing,* MIT Press, Cambridge, Mass. (1987).
2. C. Seitz, "The Cosmic Cube," *Communications of the ACM* **28**(1)(January 1985).
3. L. D. Wittie, "Communication Structures for Large Networks of Microcomputers," *IEEE Trans. on Computers* **c–30**(4)(April 1981).
4. M. E. Marhic, "Combinatorial Star Couplers for Single-Mode Fibers," *Proc. FOC/LAN '84*, pp. 175-179 (1984).
5. D. L. Baldwin, E. H. Tegge, K. Whiteleather, and S. Gilstad, "Fiber Optic Local Area Network Developments for Over 100 Ports and Approaching 1Gbps," *FOC/LAN,* (1986).
6. G. S. Christensen, "DATApipe - A High-Speed LAN," *Proc. Fifth European Fibre Optic Communications and Local Area Networks Exposition (EFOC/LAN 87),* (June 1987).
7. Y. Ofek, "The Topology Algorithms and Analysis of a Synchronous Optical Hypergraph Architecture," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, Illinois (May 1987).
8. P. K. McKinley and Y. Ofek, "Resource Sharing in a Synchronous Optical Hypergraph," *Proc. Symposium on the Simulation of Computer Networks,* (Aug. 1987).
9. P. W. Doud and K. Jabbour, "Performance Evaluation of the Spanning Multiaccess Channel Hypercube Interconnection Network," *Proc. IEEE INFOCOM '87,* pp. 1117-1125 (1987).
10. H. P. Katseff, "Flow-Controlled Multicast in Multiprocessor Systems," *Proc. IEEE Phoenix Conference on Computers and Communications,* pp. 8-13 (1987).
11. D. W. Wall, "Mechanisms for Broadcast and Selective Broadcast," Ph.D. dissertation, Stanford University, Stanford, California (June 1980).
12. L. R. Foulds and R. L. Graham, "The Steiner Problem in Phylogeny is NP-Complete," *Advances in Applied Mathematics* **3** pp. 43-49 (1982).
13. P. K. McKinley and J. W. S. Liu, "Multicast Routing in Bus-Based Computer Networks," *Proc. Computer Networking Symposium,* pp. 277-287 (April 1988).
14. P. McKinley and J. W. S. Liu, *Multicast Routing in Irregular Bus-Based Networks,* in preparation
15. A. J. Frank, L. D. Wittie, and A. J. Bernstein, "Multicast Communication on Network Computers," *IEEE Software,* (May 1985).
16. A. Frank, "Distributed Dynamic Groups on Network Computers," Ph.D. dissertation, State University of New York at Stony Brook (Dec. 1985).
17. L. Aguilar, "Datagram Routing for Internet Multicasting," *ACM SIGCOMM 84 Computer Communications Review* **14**(2) pp. 58-63 (June 1984).
18. S. E. Deering, "Internet Group Multicast Protocol," Technical Report RFC-991, SRI Network Information Center (November 1986).
19. E. Caples and C. D. Young, "Multidestination Protocols for Tactical Radio Networks," *Proc. MILCOM,* pp. 615-619 (1987).
20. Y. Lan, L. M. Ni, and A. Esfahanian, "Distributed Multi-destination Routing in Hypercube Multiprocessors," *Proc. Third Conference on Hypercube Concurrent Computers and Applications,* (Jan. 1988).
21. J. H. Saltzer, D. P. Reed, and D. D. Clark, "Source Routing for Campus-Wide Internet Transport," pp. 1-23 in *Local Networks for Computer Communcations,* ed. A. West and P. Janson,North-Holland, Amsterdam (1981).
22. D. A. Pitt, K. K. Sy, and R. A. Donnan, "Source Routing for Bridged Local Area Networks," pp. 517-530 in *Advances in Local Area Networks,* ed. K. Kummerle, J. Limb and F. Tobagi,IEEE Press, New York (1987).

FURTHER Investigated

TRANSITIVE CLOSURES GLOBALLY EXPENSIVE

A TECHNIQUE FOR ANALYZING POINTER AND STRUCTURE REFERENCES IN PARALLEL RESTRUCTURING COMPILERS

Vincent A. Guarna, Jr.

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract -- This paper describes techniques for doing dependence analysis for C programs. In particular, only those problems pertaining to aliasing due to pointers are discussed. An algorithm is developed to determine alias relationships within C programs. This algorithm is then enhanced to build dependence graphs for C looping constructs containing pointer expressions.

INTRODUCTION

Overview

Modern high-performance computing systems achieve throughput by taking advantage of parallelism realized through architectural features such as multiple processors and vector arithmetic units. These machine features are frequently utilized through the use of optimizing compilers that perform parallelization and vectorization on scientific Fortran codes. However, with the proliferation of a new genre of parallel machines running the Unix [9,12] operating system,[a] the automatic optimization of languages such as C is a subject of increasing interest.

Automatic restructuring for parallelism and vectorization is usually done by analyzing the data dependence graphs for sequential programs and ascertaining which of a number of known program transformations may be applied to introduce parallelism without changing program semantics. Dependence analysis techniques have been researched for several years, especially in the context of vectorization of Fortran programs [2,3,10,14]. However, newer languages such as C that have pointer and structure data types present different problems for dependence analysis, particularly because of new aliasing problems which can arise. This paper focuses specifically on the dependence analysis of C program constructs containing pointers and structures.

Related Work

Although work in the area of dependence analysis in the presence of pointers has not been extensive, several efforts in the area of data flow analysis have been done in the past few years.

Aho, Sethi, and Ullman [1] present heuristics for recognizing the effects of aliases created through the existence of simple pointer expressions. They also present an algorithm for collecting aliases on a global rather than a statement-by-statement basis. The resulting algorithm is conservative but fast, and should be easy to implement.

Coutant [7] presents an overview of an alias analysis technique for pointer expressions similar to that discussed in this paper. The algorithm handles structures as well as multiple levels of indirection. Coutant focuses on alias computation techniques without regard to dependence computation and subsequent parallelization. Additionally, Coutant describes an algorithm that does flow-insensitive analysis rather than point-by-point alias analysis.

[a] Unix is a trade mark of AT&T Bell Laboratories.

Weihl [13] presents an algorithm for data flow analysis of programs containing pointer references. In contrast to this paper, Weihl develops his theory with the assumption that control flow information is not available. Alias computation in Weihl is somewhat conservative because he does not differentiate between multiple levels of pointer indirection. As with Coutant's algorithm, Weihl's technique collects alias information in a flow-insensitive manner. Weihl also does not deal with the issues of arrays or structures.

No discussion of dependence analysis would be complete without referencing the work of Allen [2] and Wolfe [14]. These works concentrate on the problems of dependence analysis in Fortran programs, specifically in the area of array references. Pointer expressions are not covered.

Notation and Definitions

This section defines some of the various phraseological and notational conventions used in this paper.

Statement References. Statements within example program segments will frequently be referenced. S_i refers to statement i in a C source program fragment, and will be labeled as such in accompanying figures. For any two statements, S_i and S_j, $i < j$ implies that S_i lexically precedes S_j.

Variable References.

Definition 1-1. The Set of Referenced Variables, REF(S)

If S is a C source statement, then REF(S) is the set of all variables referenced for input by that statement. For simple assignment statements not involving pointer expressions or the C increment and decrement operators, REF(S) corresponds to the variables that appear on the right-hand side (RHS) of the assignment operator.

Definition 1-2. The Set of Generated Variables, GEN(S)

If S is a C source statement, then GEN(S) is the set of variables generated by that statement [1]. For simple assignment statements not involving pointer expressions or the C increment and decrement operators, this corresponds to the variable that appears on the left-hand side (LHS) of the assignment operator.

Definition 1-3. The KILL set

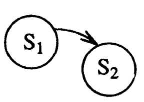
If S is a C source statement, then KILL(S) is the set of reaching definitions [1] that is killed by that statement. For single assignment statements not using the unary increment/decrement operators, KILL(S) exactly equals GEN(S).

Data Dependences.

Definition 1-4. Flow Dependence

A *flow* data dependence from S_i to S_j, denoted as $S_i \delta S_j$, exists if $\alpha \in GEN(S_i)$, $\alpha \in REF(S_j)$ for some α. S_i is the *source* of the dependence and S_j is the *sink* of the dependence. S_i must be executed before S_j. For example, $S_1 \delta S_2$ in the following FOR loop:

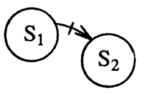
```
for (i = 1; i <= 10; i++) {
        a[i] = b[i] + c[i];
        d[i] = a[i - 1] + b[i];
}
```



Definition 1-5. Anti-Dependence

An *anti* data dependence from S_i to S_j, denoted as $S_i \bar{\delta} S_j$, exists if $\alpha \in REF(S_i)$, $\alpha \in GEN(S_j)$ for some α. S_i is the source of the dependence, and S_j is the sink of the dependence. S_i must be executed before S_j. For example, $S_1 \bar{\delta} S_2$ in the following FOR loop:

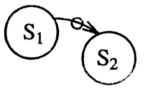
```
for (i = 1; i <= 10; i++) {
        a[i] = b[i] + c[i];
        b[i-1] = c[i] + d[i];
}
```



Definition 1-6. Output Dependence

An *output* data dependence from S_i to S_j, denoted as $S_i \delta^o S_j$, exists if $\alpha \in GEN(S_i)$, $\alpha \in GEN(S_j)$ for some α. S_i is the source of the dependence, and S_j is the sink of the dependence. S_i must be executed before S_j. For example, $S_1 \bar{\delta} S_2$ in the following FOR loop:

```
for (i = 1; i <= 10; i++) {
        a[i] = b[i] + c[i];
        a[i-1] = b[i] + d[i];
}
```



DEPENDENCES AND POINTERS

Overview

Traditional flow dependence analysis focuses on computing the intersections of GEN sets of statements with the REF sets of other statements. The existence of a non-empty intersection set between any two such statements indicates the existence of a dependence between the statements. For pointer variables, the problem is slightly more complicated. Once a pointer is assigned a valid address, the pointer may be used in two ways. One is to access the value stored in the pointer variable; this corresponds to a traditional scalar access to this variable in which no indirection is used. In the second way, the value is used to dereference the pointer one or more times (i.e. a read access is done in the process of computing the address of the target operand). Consider the following C program fragment:

```
            int     a, *p, *q;

      S1:   p = &a;
      S2:   q = p;
```

These pointer references are simply scalar references and are analyzed in the usual way [10]. The value of **p** computed in S_1 is used in S_2; therefore, a flow dependence exists from S_1 to S_2.

Consider a second example:

```
            int     a, *p;

      S1:   p = &a;
      S2:   *p = 6;
```

In this example, **p** appears on the left-hand side of the assignment in S_2; however, it appears in a dereferenced form. This program fragment has the same dependence graph as the previous one, i.e., a flow dependence exists from S_1 to S_2. Note that the appearance of a dereferenced pointer expression in a statement always constitutes a fetch of that pointer value, regardless of which side of the assignment the pointer expression appears. Because the pointer value must be fetched in order to compute the final target address of the operand, it is always a member of $REF(S_i)$.

Consider one more example:

```
            int     x, *p, **pp;

      S1:   pp = &p;
      S2:   *pp = &x;
      S3:   **pp = 3;
```

Figure 1. Example of Multiple Dereferences

In this example, the value computed in S_1 for **pp** is used in S_2. Similarly, it is used to dereference twice in S_3. Therefore, a flow dependence exists from S_1 to S_2 and from S_1 to S_3. This, however, should be intuitively insufficient. If dependences only existed from S_1 to S_2 and S_3, then S_2 and S_3 could execute in any order (or simultaneously) which is clearly not true for the given example. The reason for this is that dereferencing **pp** in S_2 yields a pointer, not a basic datum. This pointer is then dereferenced in S_3 to access the value of **x**. Therefore, a flow dependence must exist from S_2 to S_3 giving the complete dependence graph as given in Figure 2.

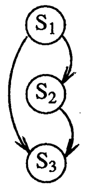


Figure 2. Dependence graph for program in Figure 1

Dependence Analysis Through Syntax Tree Matching

An accurate analysis of program dependences with pointer expressions can be obtained by building syntax trees for expressions involving pointer expressions. Before describing the use of these syntax trees, five new definitions are presented.

Definition 2-1. The Set of Trees of Referenced Variables, TREF(S)

If S is a C source statement, then TREF(S) is the syntax tree corresponding to the expression for the variables that are referenced ~~in that statement. In the case of an assignment statement,~~ TREF(S) is the abstract syntax tree for the expression to the right of the assignment operator [1].

WRONG

Definition 2-2. The Set of Trees of Generated Variables, TGEN(S)

If S is a C source statement, then TGEN(S) is the set of syntax trees corresponding to the expressions that receive new values as a result of that statement. For assignment statements, TGEN(S) is the single abstract syntax tree for the expression to the left of the assignment operator.

Definition 2-3. The TKILL set

If S is a C source statement, then TKILL(S) is the set of syntax trees for expressions that are killed by that statement. For single assignment statements not using the unary increment/decrement operators, TKILL(S) exactly equals TGEN(S).

Definition 2-4. The Set of Equivalent Trees, TEQUIV

TEQUIV is a set of unordered pairs of syntax trees (α, β) such that α aliases β (i.e. α and β represent expressions that access identical data locations). $TEQUIV_i$ contains pairs of equivalences that are valid for statement S_i. Additions to the TEQUIV set are made upon recognition of assignments to pointer expressions. For convenience, each α and β in TEQUIV is referred to as an *expression.* Additionally, for a given pair (α, β), α is the *co-alias* of β, and β is the co-alias of α.

Definition 2-5. The TDEF set

If S_i is a C source statement, then $TDEF_i$ is the set of definitions reaching that statement [1].

Using these definitions, an algorithm will be described that allows the discovery of dependences in the presence of pointer expressions. The algorithm is very similar to those algorithms used to detect dependences in non-pointer environments (e.g. Fortran). Specifically, IN and OUT sets are computed and intersected, with the resulting intersections yielding information about inter-statement dependences. The major difference is that the set elements are syntax trees rather than simple identifier entries. This is done to facilitate the recognition of aliasing situations embedded within the expressions.

The algorithm for determining dependences between C statements in a non-loop environment is given below. For simplicity, only flow and output dependences are checked. Additionally, all statements are assumed to be assignment statements. A more complete discussion about algorithm additions needed for anti-dependence discovery can be found in [8]. Dependences involving structures and loops are covered later.

Dependence Algorithm Definition. This algorithm computes dependences that exist after the analysis of a source statement S_i. Before statement S_i is analyzed, the set $TDEF_i$ is assumed to be valid. Initially, $TDEF_1$ is the null set. $TEQUIV_i$ must also be valid before statement S_i is analyzed. Initially, $TEQUIV_1$ is the null set.

STEP 1 -- Compute TREF(S_i).

First, the expression tree for the right-hand side of the assignment statement is built. With one exception, each identifier that appears as a leaf node in this tree is added to TREF(S_i). The exception is the case of identifiers in the tree that have the C "address of" (&) operator as a parent node. The "address of" operator causes the address of an identifier to be used in a computation rather than its value. Because this is a static compile-time address expression, it does not contribute to the data dependence graph for that program and so it is not entered in TREF(S_i).

Any subtrees rooted at the C dereference operator (*) are also included in TREF(S_i). Intuitively, a statement such as

x = *p;

generates two references. The first is the pointer, p, which is picked up during the frontier traversal. The second is *p, the value to which p is pointing. A similar case exists for any subtree rooted at (*).

As previously mentioned, the left-hand side of the assignment statement must also be analyzed for inclusions into TREF(S_i). Again, the syntax tree for the LHS expression is built. Similarly, identifiers on the frontier of the tree, along with subtrees rooted at (*) are considered for inclusion. This time, however, only *proper* subtrees are included in the TREF set. The subtree which is not a proper subtree represents a definition, not a reference for the given expression. For example, in the statement

p = ...

the identifier, p, appears on the frontier of the expression tree, but is not a proper subtree and, therefore, is not included in the TREF set. This is correct because p is not being referenced in this context, but is being given a value. Similarly, in the statement

*p = ...

p *is* a proper subtree and is included in the TREF set. Notice that *p is a subtree rooted at (*) and must be considered; however, it is not included because it is not a proper subtree of the LHS syntax tree.

The first pass of TREF(S_i) is now complete. The final step is to examine each element that has been put into TREF(S_i) to check for any aliases that might exist. To do this, each element of TREF(S_i) and $TEQUIV_i$ is examined. If any element of TREF(S_i) is matched with any expression in $TEQUIV_i$, the associated expression's co-alias is also included into TREF(S_i).

STEP 2 -- Compute TGEN(S_i).

TGEN(S_i) is initially set to the syntax tree for the LHS expression. Also added are the syntax trees for any expressions on the RHS that may result in a change in an identifier's value (such as a function call with the & operator applied to one or more arguments[b]). $TEQUIV_i$ is then searched to locate aliases (as in step 1). If any aliases are found, they are added to TGEN(S_i).

STEP 3 -- Compute dependences

The intersection of TREF(S_i) and $TDEF_i$ is taken. Any elements common to both sets represent flow dependence arcs into this statement based on those elements. Similarly, any elements in the intersection of $TDEF_i$ and TGEN(S_i) represent output dependence arcs into this statement based on those elements.[c]

STEP 4 -- Update $TDEF_{i+1}$

TGEN(S_i) is intersected with $TDEF_i$. If the intersection is non-null, the common elements are removed from the TDEF set.[d] Now, the union of TDEF and TGEN is taken, with the result being placed in $TDEF_{i+1}$. Also, any element in the $TDEF_i$ set that contains an element of TGEN(S_i) as a proper subtree is eliminated from $TDEF_{i+1}$ (killed).

STEP 5 -- Update $TEQUIV_{i+1}$

Each element of TGEN(S_i) is compared to the elements of $TEQUIV_i$. If any element of the TGEN set appears as a proper subtree of any of the elements in $TEQUIV_i$, the associated pair in $TEQUIV_i$ is removed and does not appear in $TEQUIV_{i+1}$. The reason for this is as follows. Each element (alias pair) in the TEQUIV set is created during the analysis of statements involving pointer assignments. For example, the statement

```
p = expr;
```

(where p is a pointer variable) creates the alias (*p, *expr) in the TEQUIV set. A definition of (or assignment to) *p, which is *not* a proper subtree, does not affect the alias relationship whereas a definition of (or assignment to) p does. In general, a definition of any of the constituent expressions that is used in the dereferencing of an alias will kill that alias.

After the necessary entries have been removed from the TEQUIV set, the LHS of the assignment statement is examined. An LHS expression of type pointer results in the creation of an alias and the pair (*LHS, *RHS) is added to $TEQUIV_{i+1}$. If the LHS expression is not a pointer, no new aliases are created.

Finally, a transitive closure is performed on the $TEQUIV_{i+1}$ set. For example, if TEQUIV includes the alias pairs (*a, *b) and (*b, *c), the pair (*a, *c) is added to the set. After this operation, $TEQUIV_{i+1}$ and the dependence analysis for statement S_i are complete.

A Simple Example. The application of the outlined procedures is shown in the context of the program in Figure 3. The analysis steps are shown in Table 1. For convenience, syntax trees are not depicted graphically; rather, they are denoted by their infix representation.

```
main()
{
        int     x, y, *p, **pp;

s1:     p  = &x;
s2:     pp = &p;
s3:     *pp = &y;
s4:     *p  = 3;

}
```

Figure 3.

Before beginning the analysis, $TDEF_1$ and $TEQUIV_1$ are both set to the null set because no definitions or equivalences are valid at the point just before statement S_1.

The final dependence graph, following complete analysis, is shown in Figure 4.

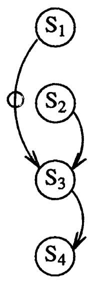


Figure 4. Dependence graph for program in Figure 3

[b] This is a pessimistic approach. Inter-procedural analysis could produce better results [5,6].

[c] The above discussions have concentrated solely on the manipulation of syntax trees generated by each statement. For inter-statement dependence computation, statement numbers must be associated with each syntax tree in the TDEF set.

[d] This is a KILL operation. For single assignment statements, the TGEN and TKILL sets are equivalent.

S_i	TREF(S_i)	TGEN(S_i)	DEPENDENCES	$TDEF_{i+1}$	$TEQUIV_{i+1}$
S_1	{}	{p}	{}	{(p,S_1)}	{(*p,x)}
S_2	{}	{pp}	{}	{(p,S_1),(pp,S_2)}	{(*p,x),(*pp,p),(**pp,x)}
S_3	{pp}	{*pp,p}	{(S_2 δ S_3),(S_1 δ^o S_3)}	{(*pp,S_3),(p,S_3),(pp,S_2)}	{(*pp,p),(**pp,y),(*p,y)}
S_4	{p}	{*p,y}	{(S_3 δ S_4)}	—	—

NOTES:

S_1: Although it may appear that **x** could be referenced by this statement, its associated node in the syntax tree has the **&** operator as a parent and is therefore not include in *TREF*. Since $TDEF_1$ is the null set, there are no dependence arcs leading to this statement. Note that the elements of the *TDEF* sets include line numbers as well as syntax trees so that inter–statement dependence arcs may easily be computed. $TEQUIV_2$ (reduced from {(*p,*(&x))}) has an entry because the LHS of S_1 is a pointer–type expression.

S_2: For $TEQUIV_3$, the last alias in the set was derived from the closure of aliases one and two (***pp** substituted for **p** in the first pair).

S_3: The generation of ***pp** is derived directly from the stmt; the generation of **p** is ascertained through the second alias in $TEQUIV_3$. The flow dependence exists because $TDEF_3$ shows a definition of **pp** in S_2 and $TREF(S_3)$ shows a use of **pp**. The output dependence is discovered because $TDEF_3$ shows a definition of **p** in S_1 while $TGEN(S_3)$ also shows a definition of **p**. $TGEN(S_3)$ contains ***pp** and **p**. $TEQUIV_3$ contains the pair **(*p,x)** of which **p** is a proper subtree and is therefore deleted. It also contains the pair **(**pp,x)** of which ***pp** is a subtree; therefore, this pair is also deleted. The pair **(*pp,p)** remains in the *TEQUIV* set. Additionally, the pair **(**pp,y)** is added as a result of the assignment statement (The LHS is a pointer expression). From the transitive closure, the last pair, **(*p,y)**, is derived.

S_4: The flow dependence is discovered because $TDEF_4$ shows a definition of **p** in S_3 and TREF(S_4) shows a use of **p**.

Table 1. Analysis Summary of Program in Figure 3

STRUCTURES AND LOOPS

Overview

This section focuses on dependence analysis for loops that operate on structures and pointers. Rather than attempting to outline a procedure that would lead to vectorization of these loops, an attempt is made to identify those aspects of C loops that can inhibit parallelization. [e]

A typical example of a loop operating on a null-terminated list structure is shown in the program in Figure 5. The goal is to determine if any dependences exist within the body of the WHILE loop that prohibit its parallelization.

In the analysis of the the WHILE loop in Figure 5, several issues warrant consideration. First, parallel execution of the loop iterations requires that all pointers to the individual instances of **struct1** be available before initiation of the parallel loop. Second, dependence analysis of the loop requires knowledge about the sequence of values that **ptr** will assume. The program in Figure 5 shows that **ptr** traverses the list threaded by the **next** fields. However, a general algorithm needs to locate the "next" and/or "previous" fields, as well as the pointers that traverse them in an arbitrary loop body. These issues are discussed in more detail in [8].

[e] Here, loop parallelization is defined as the execution of complete loop bodies by individual processors. Each processor works on an iteration in parallel until all iterations of the loop have completed. Within each processor, the entire loop body is executed serially.

Basic Loop Dependence Recognition

As with any dependence analysis operation, the OUT and IN sets are constructed and intersected. Within loop constructs however, an additional dimension is present. Before discussing references within loops, the following definition is presented.

Definition 3-1. Reference Distance

Let **P** be a pointer variable or pointer expression to some structure within a looping construct. Further, let **next** be a pointer member within the structure that threads the structures together and is used to provide a path across structures that corresponds to successive iterations of the looping construct. The *reference distance* for an expression is then the

```
main()
{
        int       x, y, z;
        struct    struct1 {
                  struct    struct1  *next;
                  int       field1;
                  int       field2;
                  int       *fieldptr;
        };

        struct    struct1  *ptr;

        ptr = (some initial value);

        while (ptr != NULL) {
                        .
                        .
                  (expr's involving ptr and struct1)
                        .
                        .
                  ptr = ptr -> next;
        }
}
```

Figure 5.

number of times the **next** field is dereferenced for that expression. For example, P -> field has a reference distance of 0; P -> next -> field has a reference distance of 1; etc. For those structures that have a **previous** field that represents structures accessed in past iterations, the reference distance for an expression is the number of times the **previous** field is dereferenced for that expression.[f] For references involving the **previous** field, the reference distance is negative. Note that references of the form P -> next have a reference distance of 0 since the **next** field is not dereferenced; rather, it is the target of the reference. Note also that references within the loop to identifiers such as global scalars that are not located within the structure being traversed have an infinite number of reference distances because they can be accessed from any iteration.

Once the reference distance for each identifier reference in a loop body is known, dependence information is easily computable. Consider two member references in a loop, r_r and r_w, where r_r is a read reference and r_w is a write reference that have reference distances d_r and d_w, respectively. Further, references r_r and r_w are located in two statements, S_r and S_w, which are not necessarily distinct and in no particular lexical order. If $d_w > d_r$, then $S_w \delta S_r$, regardless of the lexical ordering of S_w and S_r. If $d_w < d_r$, then $S_r \bar{\delta} S_w$, regardless of the lexical ordering of S_w and S_r. If $d_w = d_r$, then $S_w \delta S_r$ if S_w lexically precedes S_r and $S_r \bar{\delta} S_w$ if S_r lexically precedes S_w.

Output dependences can also be computed with the reference distance information. Consider two identifier references in a loop, r_{w1} and r_{w2}, which are both write references that have reference distances d_{w1} and d_{w2}, respectively. Further, references r_{w1} and r_{w2} are located in two statements, S_{w1} and S_{w2} which are in no particular lexical order. If $d_{w1} < d_{w2}$, then $S_{w2} \delta^o S_{w1}$, regardless of the lexical ordering of S_{w1} and S_{w2}. If $d_{w1} > d_{w2}$, then $S_{w1} \delta^o S_{w2}$, regardless of the lexical ordering of S_{w1} and S_{w2}. If $d_{w1} = d_{w2}$, then $S_{w1} \delta^o S_{w2}$ if S_{w1} lexically precedes S_{w2} and $S_{w2} \delta^o S_{w1}$ if S_{w2} lexically precedes S_{w1}.

Analysis for Loop Parallelization

In order to execute the iterations of a loop in parallel with no synchronization, dependences may only exist between statements of that loop that have expressions of equal reference distances. Pairs of references with unequal distances imply that unsynchronized accesses to the same location will occur in two or more processors. The absence of synchronization could cause nondeterministic results when compared to serial execution. Equal dependences may be ignored because they result from iterations accessing the same location within the same loop iteration. Because the goal is complete loop parallelization, equal dependences will be enforced by the serial execution of each iteration within a single processor. For completeness, however, all dependences will be computed.

Dependence Analysis for Structures and Loops

The procedure for analyzing dependences across loop iterations is the same as the tree-matching algorithm presented earlier, with two additions. The first is the analysis of structure references, the second is the analysis of the statements within loop bodies.

The analysis of structure references requires slightly more work than the analysis of scalar references because they may comprise various levels of detail. For example, in the sequence

```
struct s1 {
      struct s1 *next;
      int   x;
      int   y;
} a;

S1:  a  = ...
S2:       = a.x;
S3:  a.y = ...;
```

Figure 6.

there is a flow dependence from S_1 to S_2 because all of the fields in structure **a** are written in S_1 and one of these fields is read in S_2. However, there is no anti-dependence between statements S_2 and S_3 because there is no overlap in the memory space specified by these two references.[g]

Many of the references involving structures are of the form α->β, where α is a pointer to a structure and β is a field within that structure. These references should be stored internally as (*α).(offset,length), where "offset" is the number of bytes from the beginning of the record where the field, β, begins and "field" is the total number of bytes occupied by the referenced field. If no field is specifically referenced, then (0, sizeof(*p)) is inserted for the field. This allows more accurate analysis of

[f] The **next** and **previous** fields represent the forward and backward pointers in a doubly-linked list structure. Pointer expressions that use the **next** field are exactly analogous to array expressions in loops that use the index variable plus one. Pointer expressions that use the **previous** field are exactly analogous to array expressions in loops that use the index variable minus one.

[g] This can be complicated by the **union** construct in C, but should always be computable if the idea of overlapping memory space is used. This is essentially the same idea as the "Overlaps With" set in [7].

memory overlap conditions in the presence of union constructs and cast pointer references. Storing the offset and length instead of identifier names for fields serves two purposes. One is that Step 1 of the tree matching algorithm will not add the identifier name associated with the fields to the TREF set. Another is that the offset aids in uncovering additional aliasing problems that may be introduced with structures. An example of this will be presented later.

Maintaining pointer applications in a canonical form aids the process of alias recognition throughout the dependence analysis algorithm. Therefore, references such as p->x and p->next->x are stored as (*p).(4,4) and (*((*p).(0,4)))).(4,4), respectively, with the corresponding syntax trees shown in Figure 7.[h]

With all structure references organized in this way, the tree matching algorithm can be modified to work with structures. As originally presented, Step 1 of the algorithm traversed the frontier of the expression tree for the RHS and added all leaf nodes and subtrees rooted at "*" to the TREF set. Step 1 must also add subtrees rooted at "." to the TREF set. For the example of p->next->x, Figure 7 shows that ((*p).(0,4)) and (*((*p).0))).(4,4) are added to TREF. When this operation is complete, transitive closure of the TREF set using TEQUIV is done as before.

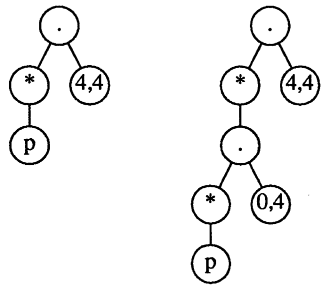


Figure 7. Syntax trees for p->x and p->next->x

Step 3 of the tree matching algorithm is also slightly different. When considering structures, dependences are recognized by locating intersections between memory spaces of fields. For example, the program in Figure 6 indicates a dependence between statements 1 and 2 because field **x**'s extent[i] (4,4) overlaps with structure **a**'s extent (0,12). With pointers and multiple structure definitions, the problem is more complex, but still solvable. Consider the program in Figure 8. The flow dependence from S_3 to S_4 can be recognized by following the augmented tree matching algorithm. At statement S_3, $TEQUIV_3$ is {(*p, x), (*q, *p), (*q, x)}. TGEN(S_3) is {((*p).(5,4)), ((*q).(5,4)), (x.(5,4))} Similarly, TREF(S_4) is {((*q).(8,4)), ((*p).(8,4)), (x.(8,4))}. Because a definition for byte 8 of structure **x** exists in statement S_3 (bytes 5, 6, 7, and 8 are defined with the offset/length pair of (5,4)) and a reference of byte 8 exists in statement S_4, statement S_4 is flow dependent on statement S_5.

[h] Offset fields have been computed with the assumption that pointers and integers require 4 bytes of internal storage.

[i] Length/offset pair.

```
main()
{
    struct s1 {
        struct s1    *next;
        char  a;
        int   b;
    } x, *p;

    struct s2 {
        struct s2    *next;
        int   c;
        int   d;
    } *q;

S1:   p = &x;
S2:   q = (struct s2 *) p;
S3:   p -> b = ...;
S4:   ...  = q -> d;
}
```

Figure 8. Example of multiple structures/ptrs

Step 3 of the tree matching algorithm also needs another addition to be able to work within the context of inter-iteration dependence analysis. As mentioned earlier, only dependences between statements that have references of unequal reference distance are of interest. This means that when intersecting the TDEF and TREF sets (for flow- and anti-dependences) and the TDEF and TGEN sets (for output dependences) the analyzer must be able to recognize overlapping accesses in the presence of multiple levels of indirection. For example, in the sequence

S_1: p -> a = ...;
S_2: ... = p -> next -> a;

the analyzer must recognize that the expression in S_2 references the same element as is defined by S_2, except in a different (future) iteration. Therefore, when comparing these syntax trees, the analyzer must be able to tolerate zero or more occurrences of "next ->" subexpressions and still find a match.

As previously mentioned, only those dependences which develop as a result of expressions of unequal reference distance are of interest in parallelization. These are the dependences that can not be honored with parallel execution of loops. Dependences occurring between expressions of equal reference distance will be enforced through the serial execution of the loop body in each processor.

As outlined, the tree matching algorithm focused solely on the analysis of straight-line code; therefore, the propagation of TDEF information was strictly linear. For looping constructs, this information must also propagate from the end of the loop body back to the beginning because the control flow of the constructs follows this behavior. Therefore, there is a cyclic flow in this information; any definition available at any point in the loop is available to any other statement in the loop. This is the same technique that is applied in any flow analysis problem [1,4].

There are two problems with the proposed dependence analysis technique. One is that the use of pointers and structures creates the possibility of circular lists. Consider the following example:

```
while (p != 0) {

/* reads and writes on fields pointed to by p */
/* assume all "flag" fields are start at 0       */

        if ((p -> flag) == 0) {
                p->flag = 1;
                p = p -> next;
        }
        else     p = p -> aux_ptr;
}
```

This loop traverses a linked list ''marking'' each structure after doing some processing on it. The program works when executed sequentially because the **flag** field can be used to control the exit from the circular list (no structure is ever processed twice in this example). In the parallel case, however, a race condition could exist. Because the list is circular, the notion of indirection distance is not valid. A write of **p -> field** and a write of **p ->** $(\text{next}\rightarrow)^n$ field will conflict for some **n**. The circularity in the data structure is probably not detectable at compile time and must be resolved through the use of assertions communicated either interactively or through source code additions.

Another inherent problem with the dependence algorithm is its complexity. Unlike languages such as Fortran where aliases may only be created at procedure call interfaces or through explicit instructions such as EQUIVALENCE, C's support for pointers allows the creation of aliases anywhere in the program. For this reason alias analysis must be combined with a detailed flow analysis to minimize the number of conservative assumptions made by the compiler. Without global analysis, for example, a pointer received as a function parameter would have to be assumed to alias all variables global to that function in order to be conservative.(j)

The potential time and space consumption for such an operation could be substantial. Transitive closure for the TEQUIV set can be an $O(N^3)$ time operation [11], where N is the number of pointer expressions appearing in the program on the LHS of assignment statements.(k) Conditional execution constructs such as IF statements and loops compound the flow analysis problem. Additional research is needed to reduce the amount of unnecessary computation for analysis and to eliminate significant sections of analysis which yield little speedup.

SUMMARY

Even in the presence of pointer expressions, dependence analysis of C programs is a solvable problem. By doing point-by-point alias computation in conjunction with program flow analysis, the number of conservative assumptions made by an optimizing restructuring compiler can be reduced.

(j) For most C programs, this requirement could be reduced to all globals of the same type as the pointer, but this is not a strict requirement.

(k) And other lvalue type operations such as call-by-reference parameters.

Because of the flexibility of the C language, alias analysis must be thorough. Alias information can be stored as pairs of abstract syntax trees, representing pairs of expressions that point to the same memory location. Additionally, the traditional use-definition information can be stored as abstract syntax trees using the alias pairs in a transitive closure operation to compute all possible references and definitions. After computing these sets of trees, sufficient information exists to discover dependence information needed for parallel restructuring.

Many issues with respect to the restructuring of C remain unanswered. First, pointers create a unique problem for dependence analysis that does not typically appear in the analysis of arrays. This is the problem of circular lists. If user data structures are linked together in some arbitrary way, the compiler has no way to determine dependence distance with certainty. In the short term, this problem can be alleviated with the insertion of programmer assertions into the source code, but this solution is suboptimal since users are prone to err. Perhaps additional analysis techniques could be found to compute this information automatically.

Second, the tree pattern matching algorithm described is likely to be time consuming (being at least $O(n^3)$). The problem is exacerbated by the fact that C applications tend to be modular. This modularity requires compilation environments to perform extensive global analysis to be effective. Additional research is needed to determine the exact cost of the alias computation algorithm. Furthermore, the cost must be weighed against the improvement seen in programs as measured by speedup. Hopefully, studies of this type will help point the way to an ''optimal'' compilation environment which can recover the majority of available parallelism with the least amount of compilation overhead.

ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under Grants No. US NSF MIP-8410110, the U. S. Department of Energy under Grant No. US DOE-DE-FG02-85ER25001, the U.S. Air Force Office of Scientific Research Grant No. AFOSR-F49620-86-C-0136, and the IBM Donation.

REFERENCES

[1] A. V. Aho, R. Sethi and J. D. Ullman. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] J. R. Allen. ''Dependence Analysis for Subscripted Variables and Its Application to Program Transformations'', Rpt. No. 82-1105, Rice University, Dept. of Mathematical Sciences, Apr., 1983.

[3] U. Banerjee. ''Speedup of Ordinary Programs'', Rpt. No. UIUCDCS-R-79-989, Ph.D. Thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, October, 1979.

[4] J. Barth. ''An Interprocedural Data Flow Analysis Algorithm,'' *Proceedings of the 4th ACM Symposium on Principles of Programming Languages* (1977), pp. 119-31.

[5] M. Burke and R. Cytron. "Interprocedural Dependence Analysis and Parallelization," *Proceedings of SIGPLAN 1986 Symposium on Compiler Construction, SIGPLAN NOTICES, 21 (7)* (1986), pp. 162-75.

[6] K. D. Cooper, K. Kennedy and L. Torczon. "The Impact of Interprocedural Analysis and Optimization on the Design of a Software Development Environment," *Language Issues in Programming Environments. Papers of the ACM SIGPLAN 1985 Symposium* (July, 1985), pp. 107-16.

[7] D. Coutant. "Retargetable High-Level Alias Analysis," *ACM Symposium on Principles of Programming Languages* (January, 1986), pp. 110-118.

[8] V. A. Guarna Jr. "Analysis of C Programs for Parallelization in the Presence of Pointers", CSRD Report No. 695, M.S. Thesis, Univ. Of Illinois, Center for Supercomputing Research and Development, Urbana, Illinois, December, 1987.

[9] B. Kernighan and R. Pike. *The Unix Programming Environment*. Prentice-Hall, 1984.

[10] D. A. Padua and M. J. Wolfe. "Advanced Compiler Optimizations for Supercomputers," *Comm. ACM 29* (December, 1986), No. 12, pp. 1184-1201.

[11] E. Reingold, J. Nievergelt and N. Deo. *Combinatorial Algorithms - Theory and Practice*. Prentice-Hall, Englewood Cliffs, N.J., 1977.

[12] University of California. *UNIX User's Manual, Reference Guide--4.2 Berkeley Software Distribution*. Computer Science Division, University of California, Berkeley, California, 1984.

[13] E. W. Weihl. "Interprocedural Data Flow Anlyisis in the Presence of Pointers, Procedure Variables, and Label Variables," *Seventh Annual ACM Symposium on Principles of Programming Languages* (1980), pp. 83-94.

[14] M. Wolfe. "Optimizing Compilers for Supercomputers", Rpt. No. UIUCDCS-R-82-1105, Ph.D. Thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, October, 1982.

INTERPROCEDURAL ANALYSIS FOR PARALLEL COMPUTING*

Zhiyuan Li Pen-Chung Yew

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
104 S. Wright Street
Urbana, Illinois 61801

Abstract — This paper presents an approach to performing interprocedural analysis for program restructuring and parallelization. Compared to previous approaches, it provides more information needed for most data dependence test schemes. It is quite effective in loop parallelization, loop restructuring, as well as array alias recognition (without array linearization). The paper also discusses the issue of recursive calls in program parallelization. It is shown that the proposed approach can handle recursive calls quite effectively.

1. Introduction

In the past few years, there has been tremendous increase in the speed of computers designed for scientific computing. One of the major factors is the improvement in the design of parallel systems. To fully capitalize on these advanced computers, an optimizing compiler should identify independent subtasks in a program which can be executed in parallel. Especially, it is very important to uncover loops whose iterations can be executed concurrently, since loops usually contain most computation in a program.

Intraprocedural analysis for parallelism has been studied for many years and has seen great progress. For programs without procedure calls, techniques for program restructuring and parallelization have been studied for more than a decade [2, 5, 21, 24, 26, 32, *etc.*]. However, handling loops with procedure calls is not well understood. When a loop has procedure calls, in the absence of information from the called procedure, the loop often has to be serialized. Conventional interprocedural analysis can provide a lot of information [3, 9, 10, 12, 14, 15, 17, 18, 20, 30, 31, 36]. Nonetheless, it ignores subscript details in array references, therefore it is usually not sufficient for loop parallelization [11, 34, 35]. In an experiment using LINPACK programs [16], we examined all the subroutines except those for complex-valued computation (which are in essence the same as their counterparts for real-valued computation). In those examined subroutines, there are 166 DO-loops, 99 of which contain procedure calls. Without information from the called procedures, these loops would have to be serialized. With conventional interprocedural analysis, only 9 procedure calls could be parallelized even after all other optimization techniques are applied, while 27 procedure calls in total could have been parallelized with more powerful techniques [28]. Moreover, a conventional interprocedural analysis does not recognize aliases as precisely as desired for program parallelization [11].

One could use *in-line expansion* [18] to eliminate procedure calls before optimizing a program for parallel machines. [19] discussed this approach. It expands the procedure calls by substituting each call statement with the body of the called procedure. After expansion, the entire program consists of only one routine. Although simple and effective, the expansion method has many well-known limitations (see e.g. [18, 28, 34, 35]).

In order to process loops with procedure calls more effectively, new techniques for interprocedural analysis are needed. Works on this subject are still quite limited [11, 28, 34, 35], and the issue of recursive calls is basically untouched. Array aliasing has not been handled except in [11] where array linearization was considered a solution. However, in many cases, linearizing an array may risk losing accuracy of analysis [29, 37]. A scheme was presented in [34, 35] to identify parallel loops with procedure calls, where programs are assumed to contain no recursion. The scheme is to analyze data dependences by checking the consistency of a set of linear inequalities which represent array references in regular statements and in called procedures. Checking such consistency is very time-consuming [27, 34, 35].

This paper is intended to present a new approach which can provide sufficient information for efficient data dependence test schemes. The approach works for programs with recursion and is quite effective for loop parallelization, loop restructuring, as well as array alias recognition (without array linearization). Our experiment [28] shows that this approach is much more efficient than the one presented in [34, 35].

2. Do-Loops and Atom Images

2.1 Parallel loops

Throughout this paper, a loop is said to be *parallel* if the iterations of the loop can be executed simultaneously without interaction. Our objective is to automatically identify parallel loops with procedure calls, some of the procedure calls may be recursive. Data dependence and alias should be analyzed for this purpose. Some loops may need restructuring (e. g. *loop distribution* and *loop interchange*) before they can be parallelized. These are discussed in [28].

The examples in figure 2.1 illustrate the nature of our problem. (Although we do not restrict ourselves to any language, the syntax and semantics in the examples are conventional and straightforward.) In Example 1, PROC1 calls PROC2 from inside a loop. PROC2 recursively calls itself. The loop in PROC1 is parallel. Example 2 shows a different pattern of array indexing. In PROC4, the parameter K changes with every call to PROC4. However, the loop in PROC3 is still parallel. In example 3, the loop in PROC5 is not parallel. In all examples, it requires interprocedural analysis to decide whether there are data dependences across the iterations of a loop. If there are no dependences across the iterations, the loop is parallel. The above observation is under the assumption that no aliases are present. Otherwise,

*This work was supported in part by the **National Science Foundation** under Grants No. US NSF MIP-8410110 and the **US Department of Energy** under Grant No. US DOE DE-FG02-85ER25001, and by a donation from the IBM Corporation. and by a donation from the CDC Corporation.

Example 1 (*i* **loop is parallel**)

```
PROCEDURE PROC1
DO i = ...
CALL PROC2(A, i)
ENDDO
RETURN

PROCEDURE PROC2(B, j)
DO K = ...
B(K, 2*j) = B(K, 2*j-1)/2
...
ENDDO
IF (...)
THEN RETURN
ELSE CALL PROC2(B,j)
RETURN
```

Example 2 (*i* **loop is parallel**)

```
PROCEDURE PROC3
DO i = ...
CALL PROC4(A, 1, i)
ENDDO
RETURN

PROCEDURE PROC4(B, K, j)
B(K, j) = B(K, j) + ...
...
ENDDO
IF (...)
THEN RETURN
ELSE CALL PROC4(B, K+1, j)
RETURN
```

Example 3 (*i* **loop is not parallel**)

```
PROCEDURE PROC5
DO i = ...
CALL PROC6(A, i)
ENDDO
RETURN

PROCEDURE PROC6(B, j)
DO K = ...
B(K, j) = B(K, j-1)/2
...
ENDDO
IF (...)
THEN RETURN
ELSE CALL PROC6(B,j)
RETURN
```

Figure 2.1

it becomes more complicated.

2.2. Data dependence test

Data dependence analysis is the most important analysis for loop parallelization and loop restructuring [2, 4, 7, 8, 38]. The commonly used schemes for data dependence test ([2, 4, 7, 8, 38]) are based on the Diophantine equations derived from the subscripts of array references. They need to know precisely the coefficients of loop indexes in each array subscript. These schemes have linear complexity with respect to the depth of loop nesting. In [34, 35], array subscripts are represented by a set of linear inequalities. To capture the global effect of array references in a procedure, those inequalities representing references to array parameters, called *global regions*, are propagated to the routines which invoke this procedure. The exact coefficients of loop indexes are not available from global regions. The only data dependence test scheme can be used in this approach is to check the consistency of a given set of linear inequalities. This scheme is very time-consuming. Theoretically, its complexity grows exponentially when the depth of loop nesting increases [13, 27, 34, 35]. Experimentally, [34] has reported an increase of testing time over those linear-time schemes by a factor of 22 to 28 on average. Moreover, this scheme does not give integral solutions which in some cases are critical. Therefore, this consistency checking scheme should only be tried when less expensive schemes failed.

When analyzing a loop without procedure calls, if subscript expressions are linear functions of loop indexes, they can be formulated precisely with a data structure called *atom* (see figure 2.2). Atoms provide the information needed in the linear-time schemes. Parafrase (an optimizing restructurer developed in University of Illinois [21, 22, 26, 32, 33, 38, *etc.*]) and its derivatives use this format for all of its test schemes. An atom is essentially a two-dimension matrix (figure 2.2). The *name* of an atom is the name of the referenced variable. If it is a read (or write) reference, the atom is called a *read* (or *write*) *atom* respectively. The matrix is empty for scalar references. For an array reference, each row of the atom corresponds to one dimension of the array, and each column (except the first one and the second one) corresponds to one of the loop indexes. The first column, marked "linear", has boolean entries which indicate if the subscript expression in a particular dimension is linear. Each entry in the second column, marked "const", is the constant term (with respect to the loop indexes) in a subscript. Each entry in the column marked "I_j" is the coefficient of index I_j in a subscript. If any non-index variable of unknown value appears in the subscript of a particular dimension of the array, in most cases, this dimension has to be marked as nonlinear and dependence is always assumed in this dimension. Note that data dependence exists between two references of an array only if it exists in every dimension.

2.3 Atom images

In order to perform data dependence test on a loop with procedure calls, atoms should be generated for variables referenced in the called procedures that can cause side effects in the calling procedure. Moreover, loop bounds in the called procedures should also be known. We introduce a data structure called *atom image* (Figure 2.3) to propagate subscript details and loop bounds from lower-level procedures. The *name* of an atom image is the name of the referenced variable declared in the called procedure. Atom images are bound to atoms when upper-level routines are analyzed. Atom images are generated only for references to formal variables and global variables which we shall call *parameters*. Without loss of generality, we assume two-level name scoping as in Fortran. Discussions can be generalized to name scoping with more levels by following the straightforward way suggested in [14]. In Fortran programs, parameters of a routine are variables in the argument list and variables in COMMON blocks. In Figure 2.3, each row with α entries corresponds to one of the dimensions of the array, and each row with β entries corresponds to one of the upper bounds of the loops (loops are normalized so that lower bounds and index steps are always 1). Each column (except the first one and the second one) corresponds to one of the loop indexes I_1, I_2, ..., I_l. The indexes are ordered by loop nesting: I_1 the outermost loop, I_l the innermost. Column "*linear*" has boolean entries which indicate whether a subscript (or a loop upper bound) is a linear expression of the loop indexes. Column "*const*" is the constant term (with respect to loop indexes) in a subscript. Each entry $\alpha_{i,j}$ denotes the coefficient of loop index I_j in dimension i of the subscript, and the entry $\beta_{i,j}$ denotes the coefficient of loop index I_j in UB_i, the upper bound of loop with index I_i. $\alpha_{i,j}$ and $\beta_{i,j}$ can be expressions which contain

```
DO I = 1, 100
DO J = 1, 100
  A(I+J, I) =
ENDDO
ENDDO
```

atom name: A

	linear	*const*	I	J
dim_1	T	0	1	1
dim_2	T	0	1	0

Figure 2.2 (a) The atom for A(I+J, I)

	linear	*const*	I_1	I_2	·	I_l
dim_1	T/F	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	·	$a_{1,l}$
dim_2	T/F	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$		$a_{2,l}$
·						
dim_d	T/F	$a_{d,0}$	$a_{d,1}$	$a_{d,2}$		$a_{d,l}$

Figure 2.2 (b) A general representation of an atom

parameters. Figure 2.4 gives an example of an atom image.

Suppose we have a loop as shown in Figure 2.5. Since subroutine SUB has a left–hand side atom image (see Figure 2.4), the CALL statement has a write atom shown in Figure 2.5. We assume array A's bound in each dimension is the same in both the calling routine and SUB. Loop upper bound K passed from SUB (now becoming N–I) should also be retained for the dependence test.

2.4 Binding atom images to atoms

Consider a call site where procedure *P2* calls *P1*. Two aspects should be considered in the binding of atom images in *P1* to atoms in *P2*. (1) For each array *A* in *P1*, its matching array *B* in *P2* should be found if *A* is a parameter. Moreover, the mapping function from an *A* element to a *B* element should be decided (for details, see [28]). (2) The expressions in an atom image (i.e. $\alpha_{i,j}$ and $\beta_{i,j}$ in Figure 2.3) should be evaluated so that the coefficients of the matching atom and the loop bounds can be decided. This is done by finding the values of parameters appearing in those expressions. If a parameter is a formal variable, its value should be the actual argument at the call site, which may be one of the following:

(1) a constant;

(2) a variable whose initial value at the call site remains constant. (Note that a variable with constant initial value may be updated inside the called procedure.)

(3) a loop index;

(4) a non–index variable of unknown value;

(5) an expression formed by factors from the above.

If a parameter is a global variable, its value may be a constant, a non–index variable of unknown value or a variable whose initial value at the call site is constant.

Generating atom images for references in regular statements (i.e. statements without procedure calls) is obvious. For call statements, atom images may need to be propagated through multiple levels. Suppose *P1* is called by *P2*, and *P2* called by *P3*. If an atom image from *P1* is bound to a variable which is a parameter in *P2*, then the reference described by the atom image from *P1* also affects *P3*. The atom image should be bound to an atom image generated in *P2*. The latter will be propagated to *P3*. Binding an atom image to another atom image is similar to binding it to an atom. However, the coefficient expressions (see Figure 2.3) in the new atom image may again contain parameters which are to be evaluated in *P3*. Note that if an atom image in *P1* is bound to an atom whose name is local to *P2*, then there is no need to generate a new atom image. This reduces some work for data dependence test in *P3*.

	linear	*const*	I_1	I_2	·	I_l
dim_1	T/F	$\alpha_{1,0}$	$\alpha_{1,1}$	$\alpha_{1,2}$	·	$\alpha_{1,l}$
dim_2	T/F	$\alpha_{2,0}$	$\alpha_{2,1}$	$\alpha_{2,2}$		$\alpha_{2,l}$
·						
dim_d	T/F	$\alpha_{d,0}$	$\alpha_{d,1}$	$\alpha_{d,2}$		$\alpha_{d,l}$
UB_1	T/F	$\beta_{1,0}$	–	–		–
UB_2	T/F	$\beta_{2,0}$	$\beta_{2,1}$	–	·	–
·						
UB_l	T/F	$\beta_{l,0}$	$\beta_{l,1}$	$\beta_{l,2}$		–

Figure 2.3 An atom image

```
SUBROUTINE SUB(I, A, K)
 DO J = 1, K
  A(I+J, I) = ...
 ENDDO
RETURN
```

atom image name: A

	linear	*const*	J
dim_1	T	I	1
dim_2	T	I	0
J	T	K	–

Figure 2.4 The atom image for A(I+J, I)

```
DO I = 1, N
 CALL SUB(I, A(1, 2), N–I)
ENDDO
```

atom name: A

	linear	*const*	I	J
dim_1	T	0	1	1
dim_2	T	1	1	0

Figure 2.5

3. Usage of Atom Images

3.1 Loop parallelization

Figure 3.1 shows a loop structure, in which we want to decide whether the loop indexed by i_m is parallel. In the loop structure, *loop body* may contain more loops which are not necessarily perfectly nested. Consider two statements, *s1* and *s2*, in *loop body*. *s2* depends on *s1* if and only if (1) there is an execution path from *s1* to *s2*; (2) *s1* has a reference *r1*, *s2* has a reference *r2*, such that the Diophantine equations which equate the subscripts in *r*1 and *r*2 have integral solutions within the loop bounds. Obviously, how the loops inside *loop body* are nested does not affect the solution of the Diophantine equations. Only the loop bounds and array subscripts have significance. [38] defined the ***direction*** of a data dependence for each loop index. Consider a data dependence from *r1* to *r2*, the dependence direction with loop index i_m may be a combination of the following:

(1) "=" direction. The Diophantine equations have a solution in which the value of i_m for *r1* equals to that for *r2*;

(2) "<" direction. The Diophantine equations have a solution in which the value of i_m for *r1* is smaller than that for *r2*;

(3) ">" direction. The Diophantine equations have a solution in which the value of i_m for *r1* is greater than that for *r2*.

A loop is a parallel loop if and only if the loop does not have a data dependence with "<" direction when all of its outer loops (if any) are only allowed to have dependence with "="

```
DO i_1 = ...
 ... ...
  DO i_m = ...

   Loop body

  ENDDO
ENDDO
```

Figure 3.1

```
DIMENSION A(100,100)
...
CALL P(A, A(1,100),50)
...
SUBROUTINE P(B,C,N)
DIMENSION B(N),C(N)
...
DO I = 1 to N
B(I) =
     = C(I)
ENDDO
```

Figure 3.2

direction. Therefore, loop nesting and statement ordering in *loop body* has no effect on whether the loop indexed by i_m is a parallel loop. For this reason, the information carried by atom images, i.e., subscripts and loop bounds, is sufficient for identifying parallel loops. In [28], it is shown that information carried by atom images is sufficient for *loop distribution*, *statement re-ordering*, as well as *loop interchange*.

3.2 Array alias recognition

Conventional alias analysis ignores subscript details in array variables. It often results in unnecessary claims for aliases and causes unnecessary dependences. Therefore, it could reduce parallelism. The example in figure 3.2 is from [11]. Conventional alias analysis would determine that B and C are aliased in *P*, as *A* is passed (by reference) to both. In fact, the *A* elements referenced by *B* and *C* in *P* do not overlap, so alias does not really exist. The problem of determining such array alias is equivalent to testing a data dependence (without restriction on dependence directions) between *B* reference and *C* reference after they are bound to *A*. [11] suggested that all arrays of multiple dimensions be linearized before data dependence test and alias analysis. Linearization certainly has its merits and sometimes may be unavoidable. Nonetheless, as [29, 37] pointed out, linearizing arrays indiscriminately may risk less precise test.

On the other hand, alias analysis could actually be performed by binding the atom images of questionable references (e.g. *B* and *C*) to atoms at higher level (e.g. atoms for *A*.) After binding, data dependence (without restriction on dependence directions) is tested between the questionable references to see if any overlap happens. For the example in figure 3.2, it is easy to test that there is no overlap between *B* reference and *C* reference. In this way, linearization could be avoided when possible. Moreover, from the information provided by atom images, arrays could still be linearized when needed.

4. Traversing the Call Graph

From the discussion in section 3, the atom image is the key to program restructuring and parallelization in the presence of procedure calls. For a given procedure *P*, once the complete set of atom images in every procedure that *P* calls is determined, atoms can be generated in *P* not only for regular statements but also for call statements. Data dependences can then be tested. Alias can be analyzed. Each loop can be examined to determine whether it is parallel. Loop distribution and loop interchange can be performed when they can be applied. (Before atoms are generated for data dependence test, some transformations on the procedure may be desired. The transformations could eliminate unnecessary dependences and could also enhance dependence tests. They were discussed in [28].) The remaining problem is to follow the call graph and generate a complete set of atom images for every procedure.

The relationship among the procedures in a program can be represented by a call graph, *G*. Each node in *G* represents a procedure and each edge indicates that a procedure calls another. If the program has recursive calls, *G* will contain cycles. Otherwise, *G* is acyclic. For an acyclic call graph, the atom image problem is simple. *G* can be traversed following the reversed direction of a ***topological sorting*** of *G* [23]. Moreover, the process of generating atom images for a procedure can be integrated into the process of optimization of the procedure. No additional visit to the procedure is needed.

In general cases, the problem is complicated by cycles in a call graph. In this section, we discuss how to obtain the complete set of atom images for every procedure, given an arbitrary call graph. Once this is achieved, individual procedures can be optimized in any order.

4.1 Notation

(1) *GEN(PROC)* denotes the set of atom images in regular statements (i.e. statements without calls) of procedure *PROC*.
(2) *OUT(PROC)* denotes the set of all atom images in *PROC*, including those from procedure calls.
(3) Φ denotes a mapping for the binding of atom images. Suppose *S* is a set of atom images from a procedure called by PROC1, Φ_{PROC_1}(S) denotes a mapping from *S* to OUT(PROC1). The value of Φ_{PROC1}(S) is the set of atom images in *PROC1* which are bound to those atom images in *S*. $\Phi_A(\Phi_B(S))$ can be written as $\Phi_A \Phi_B(S)$. $\Phi_A \Phi_B$ is called the ***product*** of Φ_A and Φ_B. Obviously, Φ is distributive, i.e. $\Phi_A(S1 \cup S2) = \Phi_A(S1) \cup \Phi_A(S2)$.

4.2 Computing the *OUT* sets

If $\{P_i \mid i = 1, 2, ..., m\}$ are procedures directly called by *P*, then

$$\mathrm{OUT(P)} = \mathrm{GEN(P)} \cup \Big(\bigcup_{i=1,m} \Phi_P(\mathrm{OUT}(P_i))\Big)$$

If we can derive such an equation for every procedure in the call graph, we shall have a system of simultaneous equations which contain recurrence and a classical flow problem is formulated. In a sense, this resembles the well-known problem of determining call effect in the classical interprocedural analysis [9, 10, 14, 30, 31]. Nonetheless, our problem is complicated not only by the parameter binding cycles but also by the subscript details. Intuitively, the solution of OUT(P) should include GEN(P) and the binding result of GEN(Q) for every procedure Q called directly or indirectly by P. However, the binding result may not be explicit due to two distinct problems caused by the binding of formal variables in recursive calls. We discuss these two problems in the following two sections.

4.2.1 Problem 1: parameters in array subscripts

First we define two classes of parameters. Suppose procedure *P1* is in a call cycle and has a parameter ***param***. If the value of ***param*** is never changed in the call cycle, we say that

```
PROCEDURE PROC1(KA, ZA)
ZA(KA) = ...
KA = KA + 1
CALL PROC2(KA, ZA)
RETURN

PROCEDURE PROC2(KB, ZB)
ZB(KB) = ...
IF (...) CALL PROC1(KB, ZB)
RETURN
```

(1) KA is explicitly updated

```
PROCEDURE PROC1(KA, ZA)
ZA(KA) = ...
CALL PROC2(KA+1, ZA)
RETURN

PROCEDURE PROC2(KB, ZB)
ZB(KB) = ...
CALL PROC1(KB, ZB)
RETURN
```

(2) KA is implicitly updated

```
PROCEDURE PROC1(KA, ZA)
INTEGER templ
ZA(KA) = ...
templ = KA + 1
CALL PROC2(templ, ZA)
RETURN

PROCEDURE PROC2(KB, ZB)
ZB(KB) = ...
....
CALL PROC1(KB, ZB)
RETURN
```

(3) Local variables in the binding chain

Figure 4.1 Recursively variant parameters

param is ***recursively invariant.*** Otherwise, ***param*** is ***recursively variant.*** A parameter can be recursively variant if the parameter is explicitly updated in a procedure in the call cycle (e.g. the parameter KA in figure 4.1(1)), or the input value of the parameter is implicitly updated through an argument binding chain (e.g. the parameter KA in figure 4.1(2)) Local variables can also be involved in a binding chain which implicitly updates a parameter recursively (e.g. figure 4.1(3)) Note that a parameter can be recursively variant even though it is not in a conventional MODIFY set. A parameter is in the MODIFY set of a procedure P only if it is explicitly updated by an assignment statement in P or any procedure called from P [9, 10, 14].

Suppose array A is a parameter in procedure $P1$, and $r1$ is a reference to A. If a recursively variant parameter is found in dimension i of the subscript of $r1$, we say that the atom image for $r1$ is ***augmented*** in dimension i. When the atom image is bound to an atom, the corresponding dimension of the atom should also be augmented. If a loop bound in the atom image has a recursively variant parameter, then the bound should be assumed conservatively. If a dimension of an atom is augmented, data dependence is always assumed in that dimension. Recall that data dependence exists between two array references only if it exists in every dimension. The *GEN* sets may contain augmented atom images.

Before traversing a call graph to generate atom images, recursively variant parameters should be recognized. They can be recognized by a flow matrix which is similar to the mapping table used in a summary analysis (see, e.g. [14]). Here we need to represent the call relationship among the procedures by a multi-graph, G_m. G_m has the same nodes as in G. However, each edge stands for a distinct call site. So there may be several edges from a calling procedure to a called procedure.

First, a complete list of formal variables from all procedures is formed. Let FV denote the list. Suppose there are N_{FV} entries in the list. The flow matrix is a N_{FV} by N_{FV} bit matrix. If there is a data flow (through assignment statements and a binding chain) in the program such that the value of a formal variable $fv1$ in $P1$ affects the value of another formal variable $fv2$ in $P2$, the entry $(fv1, fv2)$ is 1, otherwise it is 0.

The flow matrix can be obtained as follows. First, set all of the entries to 0's. Suppose $ap1$ is an actual argument at a call site where $P1$ calls $P2$ and $ap1$ is directly bound to $fv2$ in procedure $P2$. For any formal variable $fv1$ in $P1$, it is easy to determine whether the value of $fv1$ affects the value of $ap1$ (which is also the initial value of $fv2$) by conventional flow analysis. If $fv1$ does affect $ap1$, then the entry $(fv1, fv2)$ is set to 1. The transitive closure of the matrix is exactly the flow matrix we need. Using Warshall's algorithm to compute the flow matrix is straightforward, but it requires $O(N_{FV}^3)$ time [1]. Following an algorithm presented in [14], the flow matrix can be computed in $O(|E|\alpha(|E|, |N|))$ time assuming that the length of the argument list at every call site is bounded by a small constant, where $|E|$ is the number of edges in G_m, $|N|$ is the number of nodes, and α is the inverse of Ackerman's function. Its time complexity is very close to linear.

Given the flow matrix, recursive variants can be determined as follows. We first identify those caused by explicit modification. If entry $(fv1, fv1)$ is 1 and $fv1$ is in the summary modification set (which we assume available,) then $fv1$ is recursively variant. Next, we identify the other type of recursively variant formal variables. If (1) the value of $fv2$ is affected by (but not identical to) the value of $fv1$, and (2) both entries $(fv1, fv2)$ and $(fv2, fv1)$ are 1, then both $fv1$ and $fv2$ are recursively variant. If the value of $fv1$ and $fv2$ are identical, they cannot be changed by their recursive binding. Finally, if $fv1$ is recursively variant and both entries $(fv1, fv2)$ and $(fv2, fv1)$ are 1, then $fv2$ is also recursively variant.

4.2.2 Problem 2: implicit references through a binding chain

The problem can be best illustrated by an example shown in figure 4.2. Obviously, GEN(PROC1) has only one element that is the atom image for D(i,k). There is no atom image for E(i,k), since it is not a parameter. However,

```
PROGRAM MAIN
....
DO k = ...
  CALL PROC1(X, Y, Z, W, k)
ENDDO
RETURN

PROCEDURE PROC1(A, B, C, D, k)
REAL A, B, C, D, E
INTEGER i, k
DO i = ...
  D(i, k) = ...
  E(i, k) = ...
ENDDO
CALL PROC1(E, A, B, C, k)
RETURN
```

Figure 4.2

OUT(PROC1) should contain more atom images. By calling PROC1 recursively three times, we can see that OUT(PROC1) should also contain A(i,k), B(i,k) and C(i,k), each in a distinct loop originated from a distinct incarnation of PROC1. Further recursive calls will only introduce redundant atom images. Such additional atom images originated from a recursive binding chain make it difficult to obtain OUT(PROC1). Moreover, if a call graph is more complicated, traversing the graph to uncover this type of atom images can be very time consuming.

Using the mapping table defined in [14], we define an *ext–GEN* set which is an extension of a *GEN* set. A mapping table is a bit matrix of the same size as the flow matrix. However, the entry (*fv1*, *fv2*) is 1 if and only if *fv1* is eventually bound to *fv2* through a binding chain. It can be obtained as follows. First, set all of the entries to 0's. Suppose *fv1* is a formal variable in *P1* and *fv2* is a formal variable in *P2*. If *fv1* is directly bound to *fv2* at a call site where *P1* calls *P2*, the entry (*fv1*, *fv2*) is set to 1. The transitive closure of the matrix is the mapping table needed. A formal variable *fv1* in *P* is a ***feedback variable***, if there is another formal variable, *fv2*, in *P* such that the entry (*fv1, fv2*) in the mapping table is 1.

Definition (the *ext–GEN* set of procedure *P*)
1. ext–GEN(P) $\supset$ GEN(P).
2. If (i) Γ is an atom image in GEN(*P*) for a scalar formal variable, *fv1*, in *P*;
 (ii) the entry (*fv1, fv2*) in the mapping table is 1, where *fv2* is also a scalar formal variable in *P*,
then ext–GEN(P) should include the atom image for *fv2*.

3. If (i) Γ is an atom image in GEN(*P*) for an array formal variable, *fv1*, in *P*;
 (ii) there are no feedback variables in any dimension of Γ in which Γ is not augmented (*c.f.* section 4.2.1);
 (iii) the entry (*fv1, fv2*) in the mapping table is 1, where *fv2* is also an array formal variable in *P*,
then ext–GEN(P) should include an atom image whose name is *fv2*, and whose coefficient expressions are exactly the same as those in Γ.

4. If (i) Γ is an atom image in GEN(*P*) for an array formal variable;
 (ii) there are feedback variables in some dimensions of Γ in which Γ is not augmented;
 (iii) *fv1* is a feedback variable in the coefficient expressions of Γ;
 (iv) the entry (*fv1, fv2*) in 1, where *fv2* is in *P*,
then ext–GEN(P) should include an atom image whose name and coefficient expressions are the same as those of Γ, except that *fv1* in the coefficient expressions should be replaced by *fv2*;

5. If (i) Γ is an atom image in GEN(*P*) for an array formal variable which is also a feedback variable;
 (ii) there are feedback variables in some dimensions of Γ in which Γ is not augmented,
then ext–GEN(P) should include all atom images which can be obtained by combining cases 3 and 4.

The process of computing ext–GEN(P) is implied in the definition. Obviously, if $\{P_i \mid i = 1, 2, ..., m\}$ are all the procedures directly called by *P*, then

$$\text{OUT(P)} = \text{ext–GEN(P)} \cup \big(\bigcup_{i=1,m} \Phi_P(\text{OUT}(P_i))\big)$$

4.3 Traversing cyclic call graphs

The *ext–GEN* set has the following important property.

Suppose Φ_{Π} is the product of a series of Φ's, then $\Phi_P \Phi_{\Pi}$(ext–GEN(P)) $\subset$ ext–GEN(P).

It is because: (a) for any atom image in ext–GEN(P), if it is augmented in one of its dimensions, nothing would be changed in that dimension after the atom image is propagated back to P; (b) if the atom image is not augmented in a certain dimension, then the subscript in that dimension can only be changed due to feedback variables. However, by definition of ext–GEN, there must be another atom image in ext–GEN which covers the change in that dimension; (c) the definition of ext–GEN also covers the possibility that when an atom image in ext–GEN(P) is propagated back to *P*, a new atom image will be generated because it is a feedback variable.

4.3.1 Basic cycles

We first consider a basic cycle containing three procedures *A*, *B*, *C* (figure 4.3). To determine the atom images for each of the procedures in figure 4.3, the following recurrence can be derived:

$$\text{OUT(A)} = \text{ext–GEN(A)} \cup \Phi_A(\text{OUT(B)})$$
$$\text{OUT(B)} = \text{ext–GEN(B)} \cup \Phi_B(\text{OUT(C)})$$
$$\text{OUT(C)} = \text{ext–GEN(C)} \cup \Phi_C(\text{OUT(A)})$$

Without loss of generality, assume the propagation of atom images starts from node *A*, $OUT^{(0)}(A)$ = ext–GEN(A). Noting that $\Phi_A \Phi_B \Phi_C$(ext–GEN(A)) $\subset$ ext–GEN(A), $\Phi_B \Phi_C \Phi_A$(ext–GEN(B)) $\subset$ ext–GEN(B), and $\Phi_C \Phi_A \Phi_B$(ext–GEN(C)) $\subset$ ext–GEN(C), we can expand the recurrence and solve OUT(A).

$$\text{OUT(A)} = OUT^{(i)}(A)$$
$$= \text{ext–GEN(A)} \cup \Phi_A(\text{ext–GEN(B)}) \cup \Phi_A \Phi_B (\text{ext–GEN(C)})$$

Substituting OUT(A) into the original recurrence, we solve OUT(B) and OUT(C):

$$\text{OUT(B)} = \text{ext–GEN(B)} \cup \Phi_B(\text{ext–GEN(C)}) \cup \Phi_B \Phi_C (\text{ext–GEN(A)})$$
$$\text{OUT(C)} = \text{ext–GEN(C)} \cup \Phi_C(\text{ext–GEN(A)}) \cup \Phi_C \Phi_A (\text{ext–GEN(B)})$$

According to the result shown above, atom images of procedures in a basic call cycle can be obtained by traversing in the reversed direction of the cycle twice, starting from any procedure in the cycle. Whenever a procedure has been visited twice, the set of its atom images will become complete. Therefore when traversing forward to the calling procedure, the complete set of atoms of that procedure can be determined.

Figure 4.4 shows a generalized basic cycle, in which a procedure, say *B*, in the call cycle may call directly or indirectly several procedures which are not part of any cycle. We say that these procedures form a *branch* originated from *B*. It is straightforward to determine the atom images for the procedures in a generalized basic call cycle. The propagation of atom images starts from the branches. Since each branch is an acyclic subgraph, the atom images for the procedures in a branch can be determined by traversing the branch in the reversed direction of any topological sorting. After atom images are determined for all branches, the cycle can be traversed in a reversed direction starting from an arbitrary node. The traverse only needs to be iterated twice.

4.3.2 Maximum strong components in G

First, the ***branches*** of a maximal strong component (MSC) [1] can be defined analogously to the branches of a basic cycle. Then the atom images for procedures in an MSC can be determined by the following steps:

Step 1 Determine the atom images of the procedures in the branches of the MSC.

Step 2 Compute a maximal acyclic subgraph (MAS) of the MSC by first computing a DFST (depth-first spanning tree) of the MSC then deleting the back edges from the MSC to eliminate cycles.

Step 3 Let MAS^R denote the reverse of MAS (i.e. every edge in MAS^R is in the reversed direction of the corresponding edge in MAS.) Traverse MAS^R.

Step 4 Traverse the reverse of the back edges in the MSC.

Step 5 Repeat step 3 and step 4 once. The atom images in the leaves of MAS^R should be complete.

Step 6 Traverse MAS^R again. The atom images of every procedure in MSC should be complete.

By the above algorithm, every edge in MAS^R will be traversed at most three times. Every back edge in the MSC will be traversed in the reversed direction at most twice. A formal proof to the correctness of the algorithm involves tedious process of setting up the recurrence equations. So instead of presenting such a proof, we illustrate the algorithm through the following example.

Consider the call graph in figure 4.5 in which the whole graph is an MSC. Step 1 is skipped since there are no branches. In step 2, an MAS (shown by the solid edges in figure 4.5) is computed. A is the root of DSFT(MSC). In step 3, the reverse of MAS can be traversed in the order of (<C,D>, <E,D>, <D,B>, <B,A>). In step 4, the reverse of the back edges (the slim edges in figure 4.5) is traversed in the order of (<A,C>, <A,D>, <A,E>). Repeating step 3 and step 4 once, the atom images in C and E will be complete. In step 5, the atom images in every procedure will be completed in the order of (D, B, A). The correctness of the above process is ensured by the solution of the OUT sets. The recurrence in the MSC is as follows:.

$$\mathrm{OUT(A)} = \text{ext-GEN(A)} \cup \Phi_A(\mathrm{OUT(B)})$$
$$\mathrm{OUT(B)} = \text{ext-GEN(B)} \cup \Phi_B(\mathrm{OUT(C)}) \cup \Phi_B(\mathrm{OUT(D)})$$
$$\mathrm{OUT(C)} = \text{ext-GEN(C)} \cup \Phi_C(\mathrm{OUT(A)})$$
$$\mathrm{OUT(D)} = \text{ext-GEN(D)} \cup \Phi_D(\mathrm{OUT(A)}) \cup \Phi_D(\mathrm{OUT(C)}) \cup \Phi_D(\mathrm{OUT(E)})$$
$$\mathrm{OUT(E)} = \text{ext-GEN(E)} \cup \Phi_E(\mathrm{OUT(A)})$$

Expanding the recurrence, we can solve the OUT set for each procedure. For example:

$$\mathrm{OUT(B)} = \text{ext-GEN(B)} \cup \Phi_B(\text{ext-GEN(C)}) \cup \Phi_B(\text{ext-GEN(D)}) \cup \Phi_B\Phi_C(\text{ext-GEN(A)}) \cup \Phi_B\Phi_D(\text{ext-GEN(E)}) \cup \Phi_B\Phi_D(\text{ext-GEN(C)}) \cup \Phi_B\Phi_D(\text{ext-GEN(A)}) \cup \Phi_B\Phi_D\Phi_E(\text{ext-GEN(A)}) \cup \Phi_B\Phi_D\Phi_C(\text{ext-GEN(A)})$$

After step 3, the following sets have been computed: { ext-GEN(B), Φ_B(ext-GEN(C)), Φ_B(ext-GEN(D)), $\Phi_B\Phi_D$(ext-GEN(E)), $\Phi_B\Phi_D$(ext-GEN(C)), }. After step 4, the following sets have been computed: { Φ_C(ext-GEN(A)) Φ_D(ext-GEN(A)) Φ_E(ext-GEN(A)) }. In this example, step 5 is not needed. After step 6, the following sets have been computed: { $\Phi_B\Phi_D$(ext-GEN(A)) $\Phi_B\Phi_C$(ext-GEN(A)) $\Phi_B\Phi_D\Phi_E$(ext-GEN(A)) $\Phi_B\Phi_D\Phi_C$(ext-GEN(A)) }. Therefore, OUT(B) is complete after step 6.

4.4 Traversing an arbitrary call graph

First, we must find all the MSCs in the call graph, G. If we reduce each MSC into a node, the resulting graph (called the ***reduced graph***) will be acyclic. To determine the complete atom images in every procedure, G is traversed as follows.

Step 1 Determine the atom images in the procedures in the branches of every MSC.

Step 2 Number the MSCs by the reverse of a topological sorting of the reduced graph.

Step 3 Visit the MSCs following the numbering obtained at step 2. When visiting an MSC, determine the atom images in the procedures in each MSC.

5. Conclusion

We have presented an approach to perform interprocedural analysis for program parallelization. In this approach, the effect of a procedure call is captured by ***atom images***. An atom image precisely formulates the subscript details of each array reference. It allows more accurate data dependence tests to be used for more advanced program restructuring techniques. These techniques cannot be performed effectively using a conventional interprocedural analysis. We have implemented our approach to perform both parallelization and restructuring on programs with procedure calls in Parafrase [21, 22]. The results [28] show much better efficiency and effectiveness compared to other approaches such as the ***region test*** [34, 35].

In this paper, we also show that atom images can be used to recognize array aliases without linearizing the array as suggested in [11], and it also allows recursive procedural

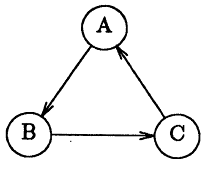


Figure 4.3 A basic call cycle

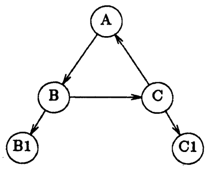


Figure 4.4 A generalized basic call cycle

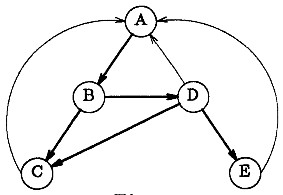


Figure 4.5

calls to be analyzed. By formulating call effects with atom images, the problem of recursion in procedure calls is reduced to finding complete sets of atom images for such procedures. We proposed algorithms to traverse a call graph and to generate those complete sets of atom images.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, Mass. 1974.

[2] J. R. Allen and K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," Dept. of Computer Science, Rice University, Houston, TX, Rice Comp TR84–9, July, 1984.

[3] F. E. Allen, "Interprocedural Data Flow Analysis," *Proceedings of the IFIP Congress*, North Holland, 1974.

[4] J. R. Allen, "Dependence Analysis for Subscripted Variables And Its Application to Program Transformations," Ph.D. Thesis, Department of Mathematical Sciences, Rice University, Houston, TX, April 1983.

[5] C. N. Arnold, "Vector Optimization on the Cyber 205," *Proceedings of the 1983 Int'l Conf. on Parallel Processing*, Aug. 1983. pp. 530–536.

[6] J. Ball, "Predicting the Effect of Optimization on a Procedure Body," *SIGPLAN Notices*, vol. 14, No. 8, 1979.

[7] U. Banerjee, "Data Dependence in Ordinary Programs," Department of Computer Sciences, University of Illinois at Urbana–Champaign, Rpt. No. 76–837, Nov. 1976.

[8] U. Banerjee, "Speedup of Ordinary Programs," Ph.D Thesis, Department of Computer Sciences, University of Illinois at Urbana–Champaign, Rpt. No. UIUCDCS–R–79–989, 1979.

[9] J. Banning, "A Method for Determining the Side Effects of a Procedure Call's," Ph.D. Dissertation, Stanford University, Aug. 1978.

[10] J. Barth, "A Practical Interprocedural Data Flow Analysis Algorithm," *CACM* Vol. 21. No. 9, Sept. 1978, pp. 724–736.

[11] M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelization," *Proc. of the ACM SIGPLAN'86 Symposium on Compiler Construction. ACM SIGPLAN Not.* 21, 7(July 1986), 162–175.

[12] D. Callahan, K. Cooper, K. Kennedy, and L. Torczan, "Interprocedural Constant Propagation," *Proceedings of the ACM SIGPLAN '86 Symp. on Compiler Construction,* SIGPLAN Notices, Vol. 21, No. 6, June 1986.

[13] P. Cousot, N. Halbwacks, "Automatic Discovery of Linear Restraints among Variables of a Program," *Conf. Record of the 5-th Annual ACM Symp. on Principle of Program Languages,* 1978.

[14] K. D. Cooper, K. Kennedy, "Efficient Computation of Flow Insensitive Interprocedural Summary Information," *Proceedings of the ACM SIGPLAN '84 Symp. on Compiler Construction,* SIGPLAN Notices, Vol. 19, No. 6, June 1984.

[15] K. Cooper, "Analyzing Aliases of Reference Formal Parameters," *Conf. Record of the 12-th Annual ACM Symp. on Principle of Program Languages,* 1985, pp281–290.

[16] J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart, *LINPACK Users' Guide,* SIAM, Philadelphia, 1979. Spring–Verlag, Heidelberg, 1976.

[17] S. Graham and M. Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis," *JACM* Vol. 23, No. 1, Jan. 1976, pp. 172–202.

[18] M. S. Hecht, *Flow Analysis of Computer Programs*, North Holland, 1977.

[19] C. A. Huson, "An In-Line Subroutine Expander for Parafrase," M.S. Thesis, Dec. 1982, Univ. of Illinois at Urbana–Champaign, Report No. UIUCDCS–R–82–1118.

[20] J. Kam and J. Ullman, "Global Data Flow Analysis and Iterative Algorithms," *JACM,* Vol. 23, No. 1, Jan. 1976, pp. 158–171.

[21] D. Kuck, R. Kuhn, B. Leasure and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Proc. of COMPSAC 80, The 4th Int'l Computer Software and Applications Conf.*, pp. 709–715, Oct. 1980.

[22] D. Kuck, R. Kuhn, D. Padua, B. Leasure and M. Wolfe, "Dependence Graphs and Compiler Organizations," *Proc. of the 8th ACM Symp. on Principles of Programming Languages* , Williamsburgh, VA, pp. 207–218, Jan. 1981.

[23] D. Knuth, *The Art of Computer Programming,* Vol. 1, Addison–Wesley, Reading, Mass. 1973.

[24] D. Kuck, "A Survey of Parallel Machine Organization And Programming," *ACM Computing Surveys*, Vol. 9, No. 1, Mar. 1977, pp. 29–60.

[25] D. Kuck, *The Structure of Computers and Computation*, John Wiley and Son, 1978.

[26] D. Kuck, "Automatic Program Restructuring for High-Speed Computations," *Proc. of CONPAR 81, Conf. on Analyzing Problem-Classes and Programming for Parallel Computing,* Nurnberg, F. R. Germany, ed. by W. Handler, June 1981 (Springer–Verlag).

[27] R. Kuhn, "Optimization And Interconnection Complexity for: Parallel Processors, Single–Stage Networks, And Decision Trees," Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana–Champaign, Rpt. No. UIUCDCS–R–80–1009, Feb. 1980.

[28] Z. Li and P. C. Yew, "Interprocedural Analysis and Program Restructuring for Parallel Programs," CSRD report No. 720, University of Illinois at Urbana–Champaign, Jan. 1988.

[29] Z. Li and P. C. Yew, "Program Parallelization with Interprocedural Analysis," Submitted for publication. April, 1988.

[30] E. Myers "A Precise And Efficient Algorithm for Determining Existential Summary Data Flow Information," Tech. Rpt. CU–CS–175–80, University of Colorado, Boulder, Colo., March 1980.

[31] E. Myers "A Precise Inter–Procedural Data Flow Algorithm," *Conf. Record of the 8-th Annual ACM Symp. on Principle of Program Languages,* 1981, pp219–230.

[32] D. Padua, D. Kuck, and D. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. Computers,* Vol. C–29, No. 9, Sept. 1980, pp. 763–776.

[33] Parafrase Analyzer Documents, CSRD, Univ. of Illinois at Urbana–Champaign, 1985.

[34] R. Triolet, "Interprocedural Analysis for Program Restructuring with Parafrase," CSRD Rpt. No. 538, University of Illinois at Urbana–Champaign, Dec. 1985.

[35] R. Triolet, F. Irigoin, P. Feautrier, "Direct Parallelization of CALL Statements," *Proceedings of the ACM SIGPLAN '86 Symp. on Compiler Construction,* SIGPLAN Notices, Vol. 21, No. 6, June 1986.

[36] M. Wegman, F. Zadeck, "Constant Propagation with Conditional Branches," *Conf. Record of the 12-th Annual ACM Symp. on Principle of Program Languages,* 1985, pp291–299.

[37] M. Wolfe and U. Banerjee, "Data Dependence for Parallelism Detection," paper prepared for publication.

[38] M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. thesis, Univ. of Illinois at Urbana–Champaign, DCS Report No. UIUCDCS- R–82–1105, Oct. 1982.

Automatic Management of Programmable Caches (Extended Abstract)

Ron Cytron (*)
Steve Karlovsky (**)
Kevin P. McAuliffe (*)

(*) IBM T. J. Watson Research Center
Computer Science Department
Yorktown Heights, New York 10598

(**) Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

We present algorithms for compiler-directed management of cache memories, where hardware does not keep such memories consistent: caches may contain discrepant values for shared variables. Our algorithms determine when a cached value must update its shared variable, and when a processor's cached value is potentially stale. Although our algorithms are presented in the context of programmable caches, the algorithms apply to a broad class of architectures where hardware does not force coherence among processors' local memories. We present algorithms and results for cache management of automatically parallelized sequential programs. We then consider optimizing the placement of cache management instructions. These optimizations apply to programs with explicitly specified cache management instructions as well as those where such instructions are automatically determined.

1.0 Introduction

As a means of obtaining increased performance, one trend in parallel architecture is to incorporate increasing numbers of processors. To cooperate in solving a problem, these processors must share data. We consider a broad class of architectures comprised of multiple processors connected to multiple memory modules (a global memory) via a multi-cycle interconnection network. Increasing the number of processors in such a system necessarily increases the average latency associated with referencing shared data. To reduce the effective memory latency, a cache can be associated with each processor. The inclusion of processor-specific caches reduces memory latency since the resulting access time is an average of the global memory access time and cache memory access time. Moreover, since the cache services a percentage of all memory requests, network traffic is diminished, thus further reducing the average latency for global memory.

The indiscriminate use of such cache memories can introduce memory coherence problems: distinct processors can view discrepant values for the same global variable. Historically, centralized or distributed hardware cross-interrogation mechanisms enforce coherence: when a processor issues a store to an address, the hardware ensures that the value in the other processors' caches is consistent with the value stored, either by invalidating the address in cache or by updating the cache with the stored value [2, 7, 13]. Maintaining coherence by hardware introduces serialization, the manifestation of which depends on the hardware implementation [22]. Incoherence can be tolerated where discrepant values for a given address are not *observable* by any processor. Although this refinement eliminates some unnecessary serialization, all global memory accesses must be tracked. For large parallel systems, maintaining coherence exclusively in hardware is prohibitive either in cost or in serialization. For this reason, the larger systems have proposed managing caches through software [14, 17, 18]

Although it might appear that the burden of managing the caches has now fallen on the programmer, this paper is devoted to showing how automatic techniques can effectively manage software-controlled caches. Because our methods are based on dependence analysis of sequential programs, our algorithms are easily incorporated into parallelizing compilers. These algorithms have been implemented in PTRAN [4, 16]; the examples shown in this paper demonstrate the results of this implementation.

In the remainder of this section, we state our assumptions and definitions with respect to software-controlled caches and we define our execution model. In Section 2.0, we provide algorithms for determining the placement of cache management instructions (cache control points). Section 3.0 describes optimizing cache management instructions. In Section 4.0, we state our conclusions and describe future work.

(**) This work was supported in part by the National Science Foundation under Grants No. US NSF MIP-8410110, the U. S. Department of Energy under Grant No. US DOE-DE-FG02-85ER25001, and the IBM Donation.

1.1 Software-Controlled Caches

We define a programmable cache as a standard cache memory where certain control mechanisms (see below) can be invoked under software control. For the present discussion, we assume a store-in cache with sufficient size (and mapping power) to avoid any *evictions*.[1] We also assume that the "line size" of the cache is the unit of storage reference (for example, a word). Thus, no data is cached without explicit reference, and no data in cache is ever replaced outside the scope of software control. In Section 4.0, we discuss the implications of evictions, store-through caches, and longer line sizes.

Unlike traditional approaches, we allow incoherence: two processors may reference the same variable, yet view discrepant values for that variable. Thus, we have a storage hierarchy with a uniform address space, referenced through global memory and caches that afford processors an efficient, albeit potentially inconsistent, view of global memory.

Each address within the global address space is marked with cacheability status:

Cacheable:	Data at the associated address can always be cached, and software will never demand that such data leave a cache. This marking is appropriate only for read-only data or data that is never shared among processors.
Temporarily cacheable:	Data at the associated address can be cached at any reference, but software controls the durations of the data's residency in cache, using the *invalidate* or *flush* instructions described below.
Non-cacheable:	Data at the associated address can never be cached. This marking is useful where the overhead associated with managing temporarily cached data becomes excessive.

Under these definitions, data enters a cache when a processor references the associated address, only if that address is marked *cacheable* or *temporarily cacheable*. Because of the store-in and eviction-free assumptions, data leaves a cache only under software control. For temporarily cached data, a processor can issue the following instructions with respect to its own cache:

Post:	Data associated with an address is copied back to global memory. The processor's cache retains its copy of the data.
Invalidate:	Data associated with an address is marked invalid. Global memory is unaffected since the cached value is *not* copied back to global memory. When the processor next references the associated address, the reference will be satisfied at global memory.
Flush:	Both of the above operations occur: data is copied back to global memory and marked invalid in the cache.

Although we define the marking and management of cached addresses in terms of individual addresses, implementation considerations have motivated architectures to consider addresses at a coarser granularity. For example, the Ultracomputer project at NYU [12] and the RP3 project at IBM [8] have considered marking cacheability status at the page or segment level as a cost-effective alternative to marking individual addresses. In the current Ultracomputer prototype [14] and in the original treatment of this problem by Veidenbaum [20], post or invalidate instructions issued by a processor apply to the entire cache associated with that processor. The RP3 allows finer grain control of cache invalidation, including line invalidate and temporarily-cacheable-data (marked data) invalidate. Such organizations invite further compile-time optimizations as described in Section 3.0.

1.2 Execution Model

Our execution model consists of a program that has been analyzed by a parallelizing compiler such as PTRAN. Although our algorithms incorporate information obtained through interprocedural analysis, the algorithms themselves examine separately each procedure of a program in turn. In the resulting program, parallelism is achieved (in part) by executing certain loops as *DOALL* loops, where the iterations of such loops can be executed by distinct processors. In general, concurrency can be generated for arbitrary parts of a program, but in this discussion we restrict ourselves to nested DOALL parallelism. The algorithms readily generalize for COBEGIN/COEND constructs.

Each time a DOALL loop is encountered, some number of processors are assigned to execute some number of iterations of that loop. When the iterations are exhausted, the processors assigned to the loop are freed. Thus, a single DOALL loop can be executed multiple times (as shown in Figure 2), but the processors assigned to the DOALL may differ for each execution.

1.3 Notation

Our algorithms use a *scheduling vector* that specifies which loops of a program potentially execute as DOALL loops. For loop i, if $SV(i)$ = 'P', then loop i is a parallel (DOALL) loop; otherwise $SV(i)$ = 'S' and loop i executes sequentially. We also require the *data dependence* graph typically computed by parallelizing compilers. We avoid a detailed discussion of data dependence [6, 9, 21], and focus instead on those aspects relevant to our work. Two statements ST_i and ST_j participate in a *flow dependence* (denoted $ST_i \delta ST_j$) if ST_i can create data that could be consumed at ST_j. Statements ST_i and ST_j participate in an *output dependence*, if both statements write to the same location and ST_j should write last (denoted $ST_i \delta^o ST_j$. Statements ST_i and ST_j participate in an *anti-dependence* (denoted $ST_i \bar{\delta} ST_j$) if ST_i reads data that can subsequently be written by ST_j.[2] For these dependence relations, ST_i and ST_j could be executed on the same or different processors. If they execute on different processors, we assume the compiler has inserted synchronization or has

(1) An eviction is the removal of a cache line by hardware for the purpose of replacement.

(2) These last two forms of data dependence can often be eliminated by renaming techniques.

sequenced the computations so that ST_i finishes before ST_j starts [10]. Synchronization, however, is insufficient to honor flow dependences where processors cache values: sometimes a processor must *post* its cached value for access by another processor. Further, we must sometimes *invalidate* stale values in a cache, so that a subsequent reference is correctly resolved in global memory. For clarity, we do not show synchronization operations in our examples.

We use the following notation to describe how processors reference global memory: Let $Write(P_i,X)$ denote a global memory write issued by processor P_i for variable (location) X. Let $Read(P_i,X)$ similarly denote a read, and let $Ref(P_i,X)$ denote an arbitrary reference (read or a write). We use "$\rightarrow$" to denote a possible sequence of such operations. For example, notation

$$Ref(P_i,X) \rightarrow Ref(P_j,X)$$

denotes that P_j might reference X after P_i.

2.0 Algorithms

We determine in three steps cacheability and associated coherence requirements.

1. Mark all variables as *temporarily cacheable*.
2. Determine where cache actions (post, invalidate, or flush) are necessary to maintain coherence, using algorithms in the ensuing sections. Given our assumptions with regard to line size and eviction, this determination is *partitionable*: cache management instructions for a given variable are not affected by such considerations for other variables. Thus, each variable can be analyzed separately; the solution over all variables is just the *union* of post and invalidation points.
3. Identify the actual cacheability of variables. For variables that require no cache action, the variables should be marked *cacheable*. Variables that are referenced only after invalidation should be marked *non-cacheable*. A variable whose behavior falls between these two extremes requires analysis to determine the profitability of caching that variable. Such analysis is beyond the scope of this paper.

2.1 Processor-Crossing Dependences

To determine cache control points, our algorithms must determine when the source and sink of a dependence potentially execute in different processors. The dependence analysis in PTRAN yields a collection of fully-refined *direction vectors* for each dependence that cannot be refuted by the decision algorithms [9]. We avoid a detailed discussion of direction vectors [21], and focus on the properties relevant to our discussion. A fully-refined direction vector can be classified as one of the following:

Loop-Carried(i)	The dependence is satisfied by sequencing the iterations of some loop *i*, where loop *i* surrounds the source and sink of the dependence [6].
Loop-Independent	The dependence is implicitly satisfied for the nested DOALL model of parallel execution.[3]

The algorithm shown in Figure 1 determines if a dependence *crosses* processors. For (*source*δ*sink*), the dependence crosses processors if *source* and *sink* potentially execute in different processors.

```
procedure CROSSES? (source, sink)

If the dependence is loop-carried (by loop i),
then
    If SV(i) = 'P'
    then return (CROSSES)
    else mark__loop = i
else mark__loop = ICLoop (source, sink)

For each loop j such that mark__loop contains loop j
and loop j contains either source or sink,
    if SV(j) = 'P'
    then return (CROSSES)

return (DOES__NOT__CROSS)
```

Figure 1. Processor-Crossing Algorithm: ICLoop is a function that returns the innermost common loop of its arguments.

The algorithm of Figure 1 conservatively assumes that a dependence can cross processors where processors are reallocated to iterations. Consider the example of Figure 2.

```
DO i=1 to N
   DOALL j=1 to M
      B(j)   =
      X(j)   =
             = B(j) + X(j)
             = B(j-1)
      A(i,j) =
             = A(i-1,j)
   ENDDOALL

    = A(i,f(i))
ENDDO
```

Figure 2. Dependences Cross

The flow dependences for A are satisfied by the outer sequential loop. However, successive executions of the inner loop cause flow dependences that potentially cross processors, since we make no assumptions about processor allocation to loop iterations. Optimization of this situation is discussed in Section 3.0.

The complexity of this algorithm is $O(\Delta)$, where Δ is the maximum depth of interval (loop) nesting. When invoked by the other algorithms of this paper, the processor-crossing algorithm effectively executes in constant time.

(3) For more general COBEGIN/COEND parallelism, such dependences are satisfied by sequencing certain statements [5].

2.2 Posting Values to Global Memory

The purpose of posting is to keep global memory up-to-date, preventing stale values from being referenced. Our fundamental observation is that the sequence:

$$Write(P_i,X) \rightarrow Read(P_j,X),\ i \neq j \qquad (1)$$

should cause P_i to post its cache value for X to global memory. The post, in addition to the requisite synchronization or sequencing, guarantees P_j references the correct value for X in global memory. Consider some procedure Q for which parallel loops have been identified. An algorithm that captures observation (1) for procedure Q is shown in Figure 3.

```
For each statement $ST_i$ that defines variable $X$,
    if $X$ is *interprocedurally live* in $Q$, and if the definition of $X$ by $ST_i$ *reaches* any exit of $Q$,
    then POST ($ST_i$, $X$)
    else consider each flow dependence $ST_i \delta ST_j$ for $X$.
        if CROSSES? ($ST_i$, $ST_j$)
        then POST ($ST_i$, $X$)
```

Figure 3. Post Algorithm: POST (ST, V) causes the reference of variable V to be posted upon completion of statement ST.

The first POST is necessary where a procedure may define variables that are used solely by other procedures. For such definitions, no flow dependences *per se* exist in the analyzed procedure. However, interprocedural information indicates whether other procedures use the defined variable, and we do not assume that such procedures execute in the same processor that defines the variable. Intraprocedural information indicates whether the definition can persist to a procedure exit. We therefore conservatively determine that such definitions must be posted to global memory for access by other procedures. The second POST is necessary for intraprocedural flow dependences. Consider the example of Figure 2 with post instructions as shown in Figure 4.

```
DO i=1 to N
   DOALL j=1 to M
      B(j)    =
      POST (B(j))
      X(j)    =
              = B(j) + X(j)
              = B(j-1)
      A(i,j) =
      POST (A(i,j))
              = A(i-1,j)
   ENDDOALL

     = A(i,f(i))
ENDDO
```

Figure 4. Example

Note that post instructions have been placed after definitions that participate in flow dependences that cross processors. For each such definition, a post instruction is generated that references the defined location. Thus, the argument to POST must be identical to the variable defined, including any subscript expressions indexing the variable. Note that the assignment to X requires no post: a processor that creates data for X will be the same processor consuming that data.

The complexity of this algorithm is $O(\Delta \times |def \rightarrow use|)$, where Δ is the maximum depth of interval nesting (effectively constant) and $|def \rightarrow use|$ is the number of def-use data flow arcs.[4]

2.3 Invalidating Cache

The purpose of invalidation is to keep cache up-to-date, preventing stale values from being referenced. Although the cache is not actually updated at invalidation time, a subsequent reference to an invalidated address demands that the cache be refreshed from global memory. Our fundamental observation is that the sequence:

$$Ref(P_i,X) \rightarrow Write(P_j,X) \rightarrow Read(P_i,X),\ j \neq i \qquad (2)$$

could cause processor P_i to have a stale value of X in its cache. In such cases, P_i must invalidate X. We present two solutions for the invalidation problem. Our first solution is conservative with respect to observation (2) above. This solution is analogous to the posting algorithm: invalidation points are determined by examining each dependence in turn. A more precise (and expensive) scheme is considered in Section 2.3.2.

2.3.1 Simple Scheme

As a first approximation to observation (2) above, consider the algorithm shown in Figure 5 as applied to some procedure Q .

```
For each statement $ST_j$ that uses variable $X$,
    if $X$ is *upwards-exposed* for procedure $Q$,
    then INVALID ($ST_j$, $X$)
    else consider each flow dependence $ST_i \delta ST_j$ for $X$.
        if CROSSES? ($ST_i$, $ST_j$)
        then INVALID ($ST_j$, $X$)
```

Figure 5. Simple Invalidation: INVALID (ST, V) causes a processor to invalidate its cache for variable V prior to executing statement ST.

The first INVALID causes a processor to invalidate its cache of a variable that could be defined by some other procedure; the invalidation occurs before the variable is accessed. Consider a dependence $ST_i \delta ST_j$ on a variable X, for which the algorithm of Figure 3 posts X after ST_i. The second INVALID of Figure 5 determines an invalidation of X before executing ST_j.

Returning to the example of Figure 4, the program shown in Figure 6 shows where invalidations occur.

(4) Flow dependences are a subset of def-use arcs.

```
DO i=1 to N
   DOALL j=1 to M
      B(j)    =
      X(j)    =
              = B(j) + X(j)
              INVALID (B(j-1))
              = B(j-1)
      A(i,j) =
              INVALID (A(i-1,j))
              = A(i-1,j)
   ENDDOALL

              INVALID (A(i,f(i)))
    = A(i,f(i))
ENDDO
```

Figure 6. Invalidations under Simple Scheme

The invalidations for this example essentially occur at the sink of dependences that caused posts in Figure 4. The complexity of this algorithm is therefore $O(\Delta \times |def \rightarrow use|)$.

2.3.2 Better Scheme

Unfortunately, excessive invalidations result from applying the algorithm of Figure 5. Returning to the example of Figure 6, the invalidate of $A(i,f(i))$ is unnecessary. Each element of A is defined at most once. Suppose a processor has a value for $A(i,f(i))$ in its cache. If that processor subsequently references that value, then the value can come from cache since no other processor could have defined the value. If the value is missing from cache, then the reference is resolved in global memory. The corresponding definitions would be posted, as determined by the algorithm of Figure 3.

In contrast to observation (2), the simple invalidation algorithm detects the more conservative situation:

$$Write(P_j,X) \rightarrow Read(P_i,X), j \neq i$$

We can eliminate invalidation where defined data can reside in at most one cache. The algorithm shown in Figure 7 captures such cases.

```
For each statement $ST_j$ that uses variable $X$,
   if $X$ is *upwards-exposed* for procedure $Q$,
   then INVALID ($ST_j$, $X$)
   else consider each flow dependence $ST_i \delta ST_j$ for $X$.
      if CROSSES? ($ST_i$, $ST_j$)
      then if $\exists ST_k$ such that ($ST_k \delta^o ST_i$ or $ST_k \bar{\delta} ST_i$ ) and CROSSES? ($S_k$, $S_i$)
      then INVALID ($ST_j$, $X$)
```

Figure 7. Better Invalidation

The second INVALID occurs where some processor may read or write the value assigned by statement ST_i. That value could then be stale at the reference by ST_j and thus require invalidation. Using the algorithms of Figure 3 and Figure 7 results in the program shown in Figure 8.

```
DO i=1 to N
   DOALL j=1 to M
      B(j)    =
              POST (B(j))
      X(j)    =
              = B(j) + X(j)
              INVALID (B(j-1))
              = B(j-1)
      A(i,j) =
              POST (A(i,j))
              = A(i-1,j)
   ENDDOALL

      = A(i,f(i))
ENDDO
```

Figure 8. Resulting Program

In essence, this better invalidation scheme determines where a processor may safely reference data that was not obtained from global memory since processor reassignment. In Figure 8, processor reassignment occurs for the inner DOALL loop. If the processor executing the outer sequential loop is assigned to an iteration of the inner DOALL loop, then the data for the use $A(i,f(i))$ may reside in cache. Although the dependence could cross processors, no stale accesses result.

As expected, the complexity of this better invalidation algorithm is worse than for the simple invalidation scheme:

$$O(|def \rightarrow use| \times (\Delta + \Delta \times (|use \rightarrow def| + |def \rightarrow def|)))$$

Where the maximum interval depth is considered constant, and where variables fewer definition than use sites, the complexity is essentially $O(|def \rightarrow use|^2)$. In practice, this complexity would be noticed only where statements define most variables. Otherwise, the observed complexity should be closer to $O(|def \rightarrow use|)$.

2.4 Flush (Post and Invalidate)

In the above discussion, posts were associated with definitions and invalidates with uses. This section examines how the two operations can be combined into a *flush* operation performed after the definition. A flush of X causes X to be posted to global memory and invalidated from the issuing processor's cache. The advantage of separating post and invalidate is that certain uses may benefit from cache accesses, even while other uses of the same variable go to global memory. In the example of Figure 8, the use of $B(i)$ can be resolved in cache while the use $B(i-1)$ must be satisfied in global memory. If the post and invalidate were combined at the definition ($FLUSH(B(i))$), then the resulting program would forfeit resolving one use of B in cache. We seek a solution where invalidates and posts can be combined without loss of cache utilization. Consider the example shown in Figure 9. Data dependence identifies a processor-crossing flow dependence for X from ST_2 to ST_4, causing the definition to be posted after ST_2 and the use to be invalidated before ST_4. No cache action is required for the dependence from ST_6 to ST_7. Thus, the invalidation inside the DOALL loop can be eliminated in favor of a flush of X after ST_2.

```
ST1  DO i=1 to N
ST2     X =
        FLUSH (X)

ST3     DOALL j=1 to N
ST4       = X
ST5     ENDDOALL

ST6     X =
ST7       = X
ST8  ENDDO
```

Figure 9. Flush Example

The algorithm shown in Figure 10 computes those definitions for which flushes do not sacrifice cache utilization.

```
do while ∃ unexamined flow dependence arcs for variable X
    Pick some arc defsite → usesite
    Initialize
        DEFSITES = {defsite}
        USESITES = {usesite}
    Compute the "closure" of the dependence; repeat
    until no changes to DEFSITES or USESITES:
        if ∃newdefsite ∉ DEFSITES such that
        newdefsite δ usesite, where
        usesite ∈ USESITES,
        then DEFSITES = DEFSITES ∪ newdefsite
        if ∃newusesite ∉ USESITES such that
        defsite δ newusesite, where
        defsite ∈ DEFSITES,
        then USESITES = USESITES ∪ newusesite
    if ∀defsite ∈ DEFSITES,
        POST (defsite, X)
      and ∀usesite ∈ USESITES,
        INVALID (usesite, X)
    then
        for each defsite ∈ DEFSITES, replace POST
        (defsite, X) with FLUSH (defsite, X)
        for each usesite ∈ USESITES, eliminate
        INVALID (usesite, X)
```

Figure 10. Flush Algorithm

With the proper data structure, the complexity of this algorithm is $O(|def \rightarrow use|)$.

3.0 Optimizations

Although the placement of cache management instructions determined by the algorithms of the preceding section is correct, the resulting programs are not necessarily optimal with respect to cache utilization or program speedup. In this section, we describe how the placement of cache management instructions can be improved. In Section 3.1, we use data flow techniques to improve cache utilization and to reduce synchronization delay for values posted to global memory. In Section 3.3 we consider how variables with similar cacheability profiles could be grouped together. Single cache instructions could then concurrently manage all members of a group. In Section 3.2, we consider how process formation and processor allocation can influence cache performance. The techniques discussed in this section are of interest for explicitly parallel as well as automatically parallelized programs.

3.1 Data Flow Motion

Here we seek to improve the placement of cache management instructions through "standard" data flow analysis. For a given region of a program, such analysis typically computes [15]:

KILL()	The set of variables for which a definition occurs along every path through the region.
PRESERVE()	The set of variables for which some path through the region contains no definition.
NODEF()	The set of variables for which no definition occurs along any path through the region.

A region that preserves *X* may or may not define *X*, but a region that kills *X* always defines *X*. Consider two processes P1 and P2 as shown in Figure 11.

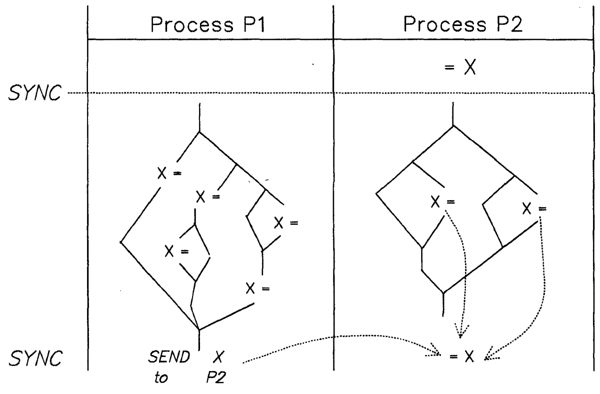


Figure 11. Problem Statement

Process P1 contains a region of code that *kills X*: every path through P1 contains some definition for *X*. The region is followed by synchronization, at which point P1 has no more updates for *X* as far as P2 is concerned. Note that such synchronization could have been explicitly specified in a parallel program. Process P2 begins by referencing *X*. This reference is either resolved in cache or causes a value for *X* to be cached in P2. Note that the algorithm of Section 2.3.2 can allow such references to be safely resolved in cache. P2 then executes a region of code that *preserves X*: some path through this region avoids defining *X*. The preserving region in P2 is followed by a use of *X* that is reached by the dependence arcs shown in Figure 11. The semantics are as follows: If the region of code in P2 defines *X*, then the use of *X* should reference the value computed by P2 (potentially in P2's cache). Otherwise, P2 fails to define *X* and the use should reference the value created by P1.

This scenario is actually a very general setting for our problem. Control flow within process P2 decides whether P2 receives its locally computed value for X or receives a value computed by some other process. When the value comes from another process, P2 must *invalidate X*: the reference to X prior to the synchronization point can result in a stale value for X in P2's cache. Similarly, control flow within P1 decides which definition of X in P1 should reach the last use of X in P2, should P2 fail to define X itself. Although the synchronization point is shown after the region that kills X, we wish to *post* X from the cache of P1 as early, yet as infrequently, as possible. This allows other processors that wait on results from P1 to proceed as soon as possible.

3.1.1 Invalidating

Applying the algorithm of Section 2.3.2 allows references prior to the synchronization point to be resolved in cache. Unfortunately, the use of X by P2 after the synchronization point would be preceded by an invalidation of X, even though some paths assign X prior to the use. We wish to invalidate X only if it has not been updated after the synchronization point. Although hardware could be developed to detect such situations, we wish to explore a software-based solution.

Given that the value for X in P2's cache is stale immediately after the *SYNC* point in Figure 11, P2 could invalidate X after the *SYNC*. Subsequent stores to X by P2 would cause P2's cache to contain the correct value for X. With respect to cache utilization, this scheme is the software equivalent of the "fast selective invalidate" scheme proposed by Cheong and Veidenbaum [11], where a bit associated with each address indicates if the address is referenced after a *SYNC* point. Such a reference causes the cache to be updated, and subsequent references are satisfied by the cache.

In general, invalidation instructions could be moved from a use site to somewhere after the synchronization point for the dependence causing the invalidation, if the following conditions hold:

1. A processor executes the invalidation instruction if the use site is reached.
2. The address(es) referenced at the use site can be generated at the invalidation point.

The first condition allows the invalidation instruction to be moved to any *dominator* of the use site in P2 (where the *SYNC* point is the final dominator).[5] The second condition is easily satisfied for scalars. For arrays, the invalidation must occur for any element that could be referenced at the use. This motivates the need for an invalidation instruction that could be applied to a group of addresses, perhaps contiguous such as those belonging to an array.

An alternative to the wholesale invalidation of such data at dominators would be to place invalidation instructions along the required paths. Consider the example of Figure 12. In Figure 12(a), invalidation always occurs for X, whereas in Figure 12(b), invalidation occurs only if the assignment to X is avoided.

There is another reason for determining the precise placement of invalidation instructions with respect to control flow. A precise invalidate (coupled with a post) corresponds to interprocessor communication, where one processor has finished updating a variable and the invalidating processor must receive the value for that variable. Once the "else" branch in Figure 12(b) is taken, X can be invalidated and the value can be requested from global memory, well in advance of the actual use of X. For processors connected via a multi-cycle interconnection network, the advance staging of such data can dramatically improve performance.

```
INVALID (X)            |
                       |
if ()                  | if ()
then  X = ...          | then X = ...
else                   | else INVALID (X)
endif                  | endif
  .                    |   .
  .                    |   .
  .                    |   .
  = X                  |   = X

       (a)                      (b)
```

Figure 12. Where to Invalidate?

This problem can be cast as a data flow problem over the control flow graph of a program. In terms of P2 shown in Figure 11, each node either kills X, preserves X, or fails to define X. The data flow problem then computes a solution that accounts for all paths through P2. The data flow values assigned at a given point E are:

VALID	All paths from the start of P2 to E contain a killing definition or invalidation of X.
NODEF	There are no definitions of X on any path from P2 to E.
PRES	Some path from the start of P2 to E defines, yet fails to kill, X.

Data flow analysis computes a solution for the entry to a node of the control flow graph. The node itself is then examined, and a value is computed for the exit(s) of that node. If *IN* is the data flow value on entry to node N and N can either $KILL(X)$, $PRESERVE(X)$, or $NODEF(X)$, then

$$OUT(N) = f(IN, Action(N))$$

follows:

$$\begin{aligned} f(VALID,Action(N)) &= VALID \\ f(PRES,KILL(X)) &= VALID \\ f(PRES,PRESERVES(X)) &= PRES \\ f(PRES,NODEF(X)) &= PRES \\ f(NODEF,KILL(X)) &= VALID \\ f(NODEF,PRESERVES(X)) &= PRES \\ f(NODEF,NODEF(X)) &= NODEF \end{aligned}$$

Where multiple paths meet at entry to a node, the meet of the data flow information is:

(5) The dominators of a node n are those nodes whose execution must have occurred if node n is executed.

Path 2

	Meet	VALID	NODEF	PRES
Path 1	VALID	VALID	VALID	PRES
	NODEF	VALID	NODEF	PRES
	PRES	PRES	PRES	PRES

Note that *PRES* is *bottom* of the meet lattice, and *NODEF* is *top*.

When the data flow problem has completed, one of two values should prevail at the use of *X* in P2:

- The value *VALID* signifies that all paths either killed *X* or could contain the appropriate invalidation instructions. Such instructions are placed on edges carrying the data flow value *NODEF*, where *VALID* meets such edges to produce *VALID*.
- The value *PRES* signifies that some path may or may not define *X*, and invalidation should be placed at some dominator of the use of *X*.

This algorithm when applied to the example of Figure 12 places invalidation on the *else* branch of the *if* statement. Consider the example shown in Figure 13.

```
if ()
then X =
else ...      *-- INVALID(X)
endif

if ()
then ....
else X =
endif

  = X
```

Figure 13. Invalidation Placement

The data flow problem places invalidation at the *else* branch of the first *if* statement. The data flow value *VALID* is subsequently propagated to the use of *X*, signifying that invalidation need not occur at a dominator of the use.

This algorithm is a *rapid* (and therefore *fast*) data flow algorithm [19]. For each variable, the algorithm takes $O(N\alpha(N))$, where N is the number of nodes in the control flow graph of a procedure. Our algorithms require the prior construction of def-use chains, which incurs similar expense.

3.1.2 Posting

In the example of Figure 11, P1 contains a region where every path defines *X*. At the end of the region, P1 should make its value for *X* available for process P2. Thus, P1 should post its value for *X* to global memory. Although the post instruction could occur at the synchronization point terminating the region, we wish to issue the post as early, yet as infrequently, as possible. The post can occur whenever we are certain P1 will make no further assignment to *X*. This allows process P2 to proceed before P1 reaches the declared synchronization point. We compute post points by solving a dataflow problem similar to *very busy expressions* [3] over the expression *X*, where

- all uses of *X* within the killing region of P1 are ignored
- the synchronization point at the end of P1 is treated as the sole use of *X*.

Posts can be placed where *X* is very busy, as shown in Figure 14.

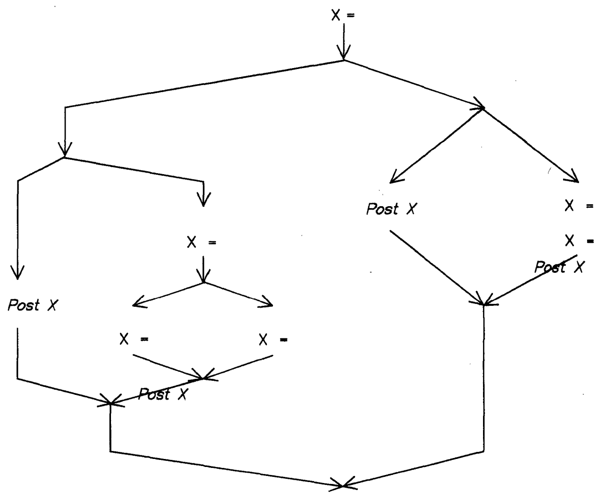


Figure 14. Post Points

3.2 Processor Allocation

Our execution model contained no assumptions with respect to the allocation of processors within or between DOALL loops. With greater supervision over processor allocation, a compiler could conceivably increase reuse of data in cache. Consider Figure 6. If for each iteration of the outer loop, iterations of the inner loop were assigned to the same processors, then the invalidations would not be necessary. As another example, consider Figure 15.

```
DOALL i= ...
   A(i) =
ENDDOALL

IF () THEN
   DOALL i= ...
      A(i) =
   ENDDOALL
ENDIF

DOALL i= ...
      = A(i)
ENDDOALL
```

Figure 15. Example for Supervised Processor Allocation

Where processors cannot be repeatedly assigned the same iterations, an invalidation must occur either at the end of the first loop or before the use of *A* in the last loop. Either invalidation prohibits the values for A computed by the first loop to remain in cache for use by the last loop. The invalidation is extraneous if the second loop does not execute. If iterations are assigned consistently to processors for the second and third loops, then the invalidation optimization algorithm of Section 3.1.1 can place invalidation inside the IF-block. The effects of invalidations can also be reduced by locality-increasing transformations such as *loop fusion* [1].

3.3 Grouping

Invalidation could benefit from a mechanism that allows a set of individual invalidates to be combined into a single *group invalidate*. Consider Figure 6. The invalidations for A are due to the cross processor flow dependences carried by the outer loop. Rather than issuing individual invalidations for each reference $A(i-1,j)$, a group invalidate for all addresses associated with A could be executed once by each processor assigned to the inner DOALL loop. This mechanism requires hardware assistance, and the optimization of group selection is beyond the scope of this paper.

4.0 Conclusion and Open Problems

The algorithms presented in this paper are sufficiently simple and fast to be implemented in parallelizing compilers. We have implemented the algorithms described in Section 2.0 in PTRAN; the effectiveness of these algorithms has yet to be determined. The following sections describe the effects of relaxing certain assumptions under which the algorithms were developed.

4.1 Line Size

We have thus far assumed that cache activity is regulated at the granularity of an individual storage reference (for example, a word). To exploit locality of reference, many systems organize cache by *lines*, where a single line contains multiple words. When a reference is satisfied by bringing a word into cache, other words associated with the referenced line are also brought into cache. Thus, words can be brought into cache without actually referencing the associated addresses. The algorithms presented in this paper assumed that if a variable is invalidated, only a subsequent reference to that variable could bring the variable back into cache. Optimizations that move an invalidation instruction away from a use are potentially incorrect where an intervening reference indirectly causes the variable to enter cache. Consider the example shown in Figure 16.

```
     P1                   P2

                       INVALID (X)
                         = Y
    X =
    POST (X)
                         = X
```

Figure 16. Line Size Problems: The reference to Y brings in a value for X if X and Y are in the same cache line.

With the invalidate of X moved away from the use, the use of X in P2 references stale data.

We feel that a strictly software solution to this problem is unreasonable: a compiler would have to know the mapping of variables to lines. We are currently investigating a solution involving hardware assistance. In particular, we postulate the usefulness of an *invalidate if not referenced* instruction. Such an instruction would invalidate data that was prefetched into cache due to its proximity to an actually referenced variable.

For performance considerations, invalidation should be localized within a cache line (applied to specific words) rather than invalidating the entire line. This requires residency bits for each unit of storage reference. To lessen hardware costs, residency could be maintained on a word basis: invalidating a byte would invalidate its associated word.

4.2 Store-in

Although our algorithms were developed for a store-in cache, the techniques also apply to a store-through cache. Obviously, a compiler need not post values to global memory for a store-through cache. However, store-through caches can degrade performance through increased network traffic. In particular, reduction of network traffic through optimized post instructions as considered in Section 3.1.2 is appropriate only for store-in caches.

4.3 Evictions

We have thus far assumed that data leaves a cache only under software control; however, caches typically use a hardware eviction policy. Although our algorithms are still correct, eviction beyond software control raises the following issues:

- Cache management instructions for one variable may affect the cache behavior of other variables. Strictly speaking, our assumptions as to the *partitionability* of cache management problems no longer hold. However, the actual mapping of addresses to cache locations cannot be considered at compile-time (for example, formal parameters).
- The optimization of post instructions involves holding onto cached data until a processor has finished modifying data at a given address. Where such data is prematurely evicted, optimization may suffer and network traffic may be increased.

Thus, adding eviction to our cache model results in potentially decreased performance. The above considerations suggest that even eviction should enter the realm of software control. Where the compiler determines certain data non-evictable, new data cannot cause the eviction of such data until the data is subsequently marked evictable. Cacheable data that conflicts only with non-evictable data would not enter cache on reference.

5.0 Acknowledgements

We thank Fran Allen and Michael Burke for their comments on this work.

Bibliography

1. W. A. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations. *IEEE Trans. on Computer*, C-30(5):341-356, May 1981.

2. Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. Scalable Directory Schemes for Cache Coherence. *Proceedings of the 15th International Symposium on Computer Architecture*, 1988.

3. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

4. Fran Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An Overview of the PTRAN Analysis System for Multiprocessing. *Proceedings of the 1987 International Conference on Supercomputing, Springer-Verlag*, Athens,Greece, 1987. To appear in a special issue of the Journal of Parallel and Distributed Computing.

5. Fran Allen, Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, and Vivek Sarkar. A Framework for Determining Useful Parallelism, IBM T.J. Watson Research Center, July 1988. ACM International Conference on Supercomputing '88.

6. Randy Allen and Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491-592, October 1987.

7. James Archibald and Jean-Loup Baer. An Economical Solution to the Cache Coherence Problem. *11th Int. Symp. on Comp. Arch.*, pages 355-362, 1984.

8. W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 Processor-Memory Element. *Proc. 1985 International Conference on Parallel Processing*, pages 782-789, 1985.

9. Michael Burke and Ron Cytron. Interprocedural Dependence Analysis and Parallelization. *Proceedings of the Sigplan '86 Symposium on Compiler Construction*, 21(7):162-175, July 1986.

10. Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, and David Shields. On the Automatic Generation of Useful Parallelism: A Tool and an Experiment, IBM T.J. Watson Research Center, July 1988. ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems.

11. Hoichi Cheong and Alex Veidenbaum. Stale Access Detection and Cache Coherence Enforcement Using a Flow Analysis Approach. *Proceedings of the 1988 International Conference on Parallel Processing*, 1988.

12. Jan Edler, Allan Gottlieb, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, Marc Snir, Patricia J. Teller, and James Wilson. Issues Related to MIMD Shared-memory Computers: the NYU Ultracomputer Approach. *Conference Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 126-135, Boston, Massachusetts, 1985.

13. James Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. *The 10th Int. Symp. Comput. Arch.*, pages 124-131, June 1983.

14. Allan Gottlieb. An Overview of the NYU Ultracomputer Project. in J. Dongarra, editor, *Experimental Parallel Computing Architectures*, North-Holland, 1987. Formerly Ultracomputer Note #100, Courant Institute of Mathematical Sciences, New York University (1986).

15. Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, Inc., 1977.

16. Steven R. Karlovsky. Automatic Management of Programmable Caches: Algorithms and Experience, Center for Supercomputing Research and Development, Urbana, Illinois. 1988. Master's thesis in progress.

17. David J. Kuck, Edward S. Davidson, Duncan H. Lawrie, and Ahmed H. Sameh. Parallel Supercomputing Today and the Cedar Approach. *Science*, 231:967-974, February 1986.

18. G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. *International Conference on Parallel Processing*, pages 764-771, 1985.

19. Barry K. Rosen. Monoids for Rapid Data Flow Analysis. *Siam Journal of Computing*, 9(1):159-196, Feburary 1980.

20. Alex Veidenbaum. A Compiler-Assisted Cache Coherence Solution for Multiprocessors.. *International Conference on Parallel Processing*, pages 1029-1036, August 1986.

21. Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*, PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois 1982. Report No. UIUCDCS-R-82-1105.

22. W. C. Yen, D. W. L. Yen, and K.-S. Fu. Data Coherence Problems in a Multicache System. *IEEE Transactions on Computers*, C-34:56-65, January 1985.

ALGORITHMS FOR STATIC TASK ASSIGNMENT AND SYMMETRIC CONTRACTION IN DISTRIBUTED COMPUTING SYSTEMS

Virginia M. Lo
Dept. of Computer and Information Science
University of Oregon
Eugene, OR 97403
lo@cs.uoregon.edu

Abstract -- In this paper, we look at the mapping problem, which was posed within the domain of parallel processing, and we redefine that problem for use in distributed computing systems whose underlying communication medium is a broadcast medium such as ethernet. We describe an efficient algorithm which can be utilized to find optimal assignments of tasks to processors for a wide variety of distributed algorithms when *symmetric contraction* of the algorithm is necessary. We also describe a heuristic algorithm for use in finding suboptimal assignments of tasks to processors for arbitrary distributed computations. Both algorithms model the mapping problem as the Graph Partitioning Problem. Our algorithms utilize an efficient algorithm for finding maximum weight matchings to find an assignment of tasks to processors which minimizes the total interprocessor communication cost while meeting a constraint on the number of tasks assigned to each processor.

1. Introduction

Researchers in the area of distributed computing are actively seeking ways to take advantage of the potential of these systems for parallel computing. The last few years have seen intensification of efforts in the design of large-grained, loosely coupled parallel algorithms and in the design of systems software (operating systems, compilers) which support parallel computation. One problem that occurs in both distributed and parallel systems is the problem of assigning tasks in a distributed (parallel) computation to the processors in a distributed (parallel) system. This problem has been referred to as the **static task assignment problem** and also as the **mapping problem**, with the former appellation more commonly used in the distributed computing community and the latter terminology more commonly used in the parallel processing community. For both types of systems, the goals of task assignment include reducing interprocessor communication, load balancing, and parallelism; indeed, many approaches to the solution of this problem can be applied successfully to either type of system. However, there are differences in the architectures of distributed systems and parallel processors that affect the design of task assignment algorithms.

In this paper, we look at the mapping problem, which was posed within the domain of parallel processing, and we define an analogous problem for distributed computing systems whose underlying communication medium is a broadcast medium such as ethernet. We demonstrate that in such a system, an appropriate goal for task assignment algorithms is minimization of the total interprocessor communication costs while meeting a constraint on the number of tasks assigned to each processor. We describe an efficient algorithm which can be utilized to find optimal assignments of tasks to processors for a wide variety of distributed algorithms when *symmetric contraction* of the algorithm is necessary. We also describe a heuristic algorithm for use in finding suboptimal assignments of tasks to processors for arbitrary distributed computations. Both algorithms model the task assignment problem as the Graph Partitioning Problem. Our algorithms utilize an efficient algorithm for finding maximum weight matchings to find an assignment of tasks to processors.

We consider only distributed systems which consist of a collection of homogeneous multiple computers, each with local CPU, memory, and other hardware resources, and which communicate through an ethernet bus. A distributed computation or task force consists of a set of communicating tasks to be assigned to processors in the distributed system. We utilize the graph theoretic model [16] for these computations in which each task is modeled as a node in the graph and communicating tasks are connected by an edge whose weight equals the communication cost incurred if the tasks are assigned to different processors. We assume that communication between tasks assigned to the same processor is negligible.

In section 2 we describe the problem in more detail and discuss its relation to the contraction problem from parallel processing. In section 3 we present our algorithms as well as simulation results for the heuristic Algorithm H. The last section contains conclusions and a discussion of further work in this area.

2. Static Task Assignment and Contraction in Ethernet-based Distributed Systems

The general problem we wish to address is the assignment of tasks to processors in order to minimize interprocessor communication while meeting constraints on the number of tasks assigned to each processor. This statement of the task assignment problem [1] is useful in two slightly different contexts: (1) for initial assignment of tasks to processors (static task assignment) and (2) for assignment of tasks to fewer processors than the distributed algorithm was initially designed for, either at the time of initial assignment or dynamically at execution time due to node failure, node withdrawal, or phase transitions in the distributed computation. Node withdrawal occurs in distributed systems comprised of a network of personal workstations. At any time, one of the workstations may become unavailable to the distributed computation, either because the 'owner' withdraws it from the pool or because the load on that workstation becomes heavy.

These problems are similar to the mapping problem posed by [2] for parallel processoring systems. In parallel processors, the point-to-point nature of the communication network offers a multiplicity of interconnection options - hypercube, meshes, shuffle-exchange networks, cube-connected cycles, trees, etc. and the potential for parallel (non-interfering) communication on disjoint paths through the network. As a result, the mapping problem involves two phases, *contraction* and *layout*. Contraction involves reducing the graph G which represents the parallel algorithm to a smaller graph G' in the same family, causing several task nodes

[1] Several different optimality criteria have been studied in the graph theoretic approach to static task assignment including minimization of the total sum of execution and communication costs [11, 13, 15, 16], minimization of IPC followed by load leveling [5], and sum-bottleneck optimization [15].

in G to become associated with one node in the reduced graph G'. (Thus, if G is a tree, it must be contracted to a smaller tree G'.) In the layout phase, the reduced graph is then mapped to the interconnection hardware, with at most one task node per processor, taking into account the mapping of edges in the computation graph to edges in the processor network [2]. At execution time, the tasks assigned to a given processor are multiplexed on that processor. In an ethernet-based system, there is only one communication pathway, and all interprocessor communication competes for the same resource. As a result, the layout phase is not relevant for distributed systems (i.e., it is not necessary to map computation edges to network edges) nor is it necessary to maintain any specific interconnection structure in the contracted graph.

In addition, the criteria evaluating contractions in a distributed system is different than that proposed for parallel processing systems. In the latter systems, evaluation of contraction algorithms utilizes one or more of the following metrics: [2]

- the average number of tasks and edges from G absorbed into one node of G',
- the maximum number of tasks and edges from G absorbed into one node of G',
- the average number of edges of G mapped to an edge in G',
- the maximum number of edges of G mapped to an edge in G'.

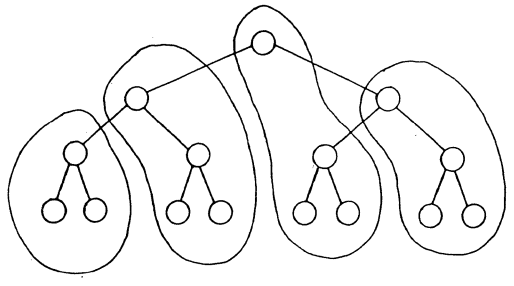
These objective functions, particularly the latter two objective functions, are useful when more than one communication link exists and communication can occur independently and in parallel on these links. In ethernet systems, a more appropriate metric with respect to communication is the total sum of interprocessor communication costs incurred by an assignment. By minimizing this quantity, the overall overhead of IPC and also the contention for ethernet is minimized thereby improving the response time of the distributed computation. However, it is a well-known fact that minimization of IPC conflicts with the goals of load balancing and parallelism. We compensate for these needs by constraining the number of tasks assignable to each processor. When many tasks are assigned to one processor they contend for the resources of that processor and incur overhead due to process switching, management of shared buffers, etc. Bounding the number of tasks per processor contributes to load balancing by limiting contention of this nature.

Thus, in an ethernet-based distributed system, task assignment requires contraction of the computation task graph to a reduced graph in order to minimize the total interprocessor communication costs while maintaining a bound on the number of tasks assigned to each processor. Henceforth, we restrict our discussion to the context of ethernet-based distributed systems and we will use the terms *task assignment, mapping,* and *contraction* interchangeably.

Because we seek different goals for contraction in distributed systems, contraction techniques devised by parallel processing researchers may not be appropriate for our problem. For example, Berman's technique of *truncation* for complete binary trees yields an assignment with total IPC cost of 12 whereas an optimal assignment according to our criteria has cost equal to 3 (see Figure 1 below).

Finally, we introduce the notion of *symmetric contraction* for distributed and parallel algorithms which are regular in structure. These algorithms consist of a number of **identical** tasks that operate on data that has been partitioned among the tasks.

[2] These performance metrics are based on the assumption that $c_{ij} = 1$ for all i,j; these metrics can be extended in a natural way for arbitrary c_{ij}.


(a) Contraction to Minimize IPC (IPC = 3)

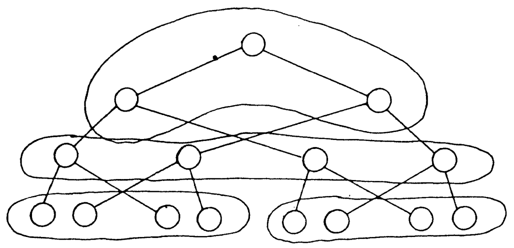

(b) Berman Snyder Contraction (IPC = 12)

Figure 1: Two Contractions of A Binary Tree
(All edges assumed to have $c_{ij} = 1$)

When contraction is performed, it is necessary to merge tasks in a symmetric fashion in order to preserve parallelism. More specifically, if a distributed algorithm consisting of k identical tasks is to be contracted for assignment to fewer than k processors, it is necessary to contract an equal number of tasks to each processor. Because the tasks on a given processor are multiplexed, an unequal contraction is undesirable because it constrains all processors to the speed of the slowest processor in the system. This necessity for symmetric contraction has been noted in [14] and in our own survey of distributed and parallel algorithms. We shall see that this fact provides compelling motivation for the use of Algorithm M (described below) to find optimal contractions of distributed computations.

3. Description of the Algorithms

We have developed two algorithms for the assignment of tasks to processors whose goals is minimization of total IPC under a common constraint on the number of tasks per processors. Algorithm M finds optimal assignments in polynomial time for a restricted group of distributed computations. Algorithm M is ideally suited for contraction of computations with the regular structure described above and for the assignment of "small" computations in a "big" distributed system. We illustrate its application to a distributed algorithm for the simplex method of linear programming. Algorithm H is an efficient heuristic which finds possibly suboptimal assignments for arbitrary distributed computations. Simulation results show the performance of this algorithm to be good, yielding an optimal assignment in 81.1% of the cases simulated.

3.1. Equivalence to the Graph Partitioning Problem

We restrict our attention to homogeneous systems with n identical processors. Let $P = \{p_1, p_2, ..., p_n\}$ be the set of n processors, $T = \{t_1, t_2, ..., t_k\}$ be a set of k communicating tasks (i.e., a distributed computation) to be assigned to the processors. Let c_{ij} be the cost of communication between tasks t_i and t_j if they are assigned to different processors. Interprocess communication cost is assumed to be negligible when communicating tasks are assigned to the same processor. Let B, $\lceil \frac{k}{n} \rceil \leq B \leq k$, be a common bound on the maximum number of tasks allowed on each processor. We define an **optimal assignment** as one which minimizes the total interprocessor communication costs incurred under the constraint that $k_q \leq B$ for all processors p_q, $1 \leq q \leq n$, where k_q is the number of tasks assigned to processor p_q.

The task-processor system described above can be modeled as a graph $G = (V,E)$ in which each task is represented as a vertex in V. An edge is constructed for each pair of communicating tasks and given a weight equal to the communication cost c_{ij}. The problem of finding an assignment of tasks to processors which minimizes IPC under a constraint on the number of tasks per processors is equivalent to the Graph Partitioning Problem with all node weights equal to one.

Graph Partitioning Problem: Given Graph $G = (V,E)$, weights $w(v)$ for each $v \in V$ and $l(e)$ for each $e \in E$, and positive integers B and J, find a partition of V into disjoint sets $V_1, V_2, \cdots V_n$ such that $\sum_{v \in V_i} w(v) \leq B$ for $1 \leq i \leq n$ and such that if E' contained in E is the set of edges that have their two endpoints in two different sets V_i, then $\sum_{e \in E'} l(e) \leq J$.

The Graph Partitioning Problem and the Graph Partitioning Problem with all vertex weights equal to one have been shown to be *NP-complete* [8]. Thus, our task assignment problem is also *NP-complete*.

3.2. Algorithm M, An Optimal Algorithm for Task Assignment

Algorithm M can be used to find optimal assignments in polynomial time when the number of tasks is less than or equal to twice the number of processors and when each processor may be assigned at most two tasks. These constraints may sound rather limiting at first, but we show that there exist many distributed computations for which these constraints hold. Algorithm M utilizes a polynomial time algorithm for finding a *maximum weight matching* in graphs. An algorithm of complexity $O(ke \log k)$ where e is the number of edges and k the number of nodes in the network is described in [7].

We first prove that for systems in which the number of tasks is less than or equal to twice the number of processors and in which each processor may be assigned at most two tasks, an optimal solution can be found in polynomial time. This proof involves two parts: (a) construction of a maximal matching in a graph corresponding to the task assignment problem, and (b) proof that a maximal matching yields an assignment which minimizes IPC while meeting the constraint of at most two tasks per processor.

Theorem 1: Consider a system with n identical processors and with the number of tasks $k \leq 2n$. Let $B = 2$ be the maximum number of tasks allowed on each processor. Then an assignment which minimizes total interprocessor communication costs under the constraint of at most 2 tasks per processor can be found in polynomial time.

Proof:

(a): Construct a graph G with a node representing each task and an edge between each pair of task nodes t_i and t_j with weight c_{ij}. We note from graph theory [9] that a *matching* in such a graph is a set of edges in which no two edges have a node in common and the *weight of the matching* is the sum of the weights of those edges that are in the matching. Furthermore, a *maximum weight matching* for G is one whose weight is maximum among all matchings for G.

A matching can be used to define an assignment of tasks to processors with at most two tasks per processor as follows:

(1) Let each pair of tasks t_i and t_j connected by an edge e in the matching be assigned to a distinct processor p_q.

(2) If there exist tasks not connected by an edge in the matching, arbitrarily arrange them in pairs and assign these pairs to distinct processors p_q such that no other tasks are assigned to p_q.

(3) If a single task remains unassigned, assign it to any processor p_q such that no other tasks are assigned to p_q.

Since the number of tasks $k \leq 2n$, there will be a sufficient number of processors to perform the above assignment. Because of the way the assignment is made, no processor will be assigned more than two tasks.

(b): We now prove that a maximum weight matching corresponds to an assignment which minimizes the total interprocessor communication costs under the constraint that no processor is assigned more than two tasks. Let f be an assignment of tasks to processors and let

$$C_f = \sum_{f(t_i) \neq f(t_j)} c_{ij} \text{ and}$$

$$\overline{C}_f = \sum_{f(t_i) = f(t_j)} c_{ij}$$

In other words, C_f is the total IPC incurred by assignment f, and $\overline{C}_f$ is the sum of communication costs between tasks assigned to the same processor. Then

$$C_{TOT} = \sum_{1 \leq i,j \leq k} c_{ij} = C_f + \overline{C}_f$$

Since C_{TOT} (the grand total of all communication costs on all edges in the graph) is fixed over all assignments, an assignment which minimizes C_f also maximizes $\overline{C}_f$.

Now consider a maximum weight matching for G and construct an assignment f from the matching as described above. We will show that the weight of the maximum weight matching is precisely equal to $\overline{C}_f$, the sum of the communication costs on edges between tasks assigned to the same processor. Each edge in the maximum weight matching has a weight c_{ij} which contributes to the sum $\overline{C}_f$ since the two tasks t_i and t_j are assigned to the same processor by step (1). Each pair of tasks selected by step (2) above has a weight c_{ij} which equals 0. (If not, that edge could be added to the maximum weight matching to produce another matching with greater weight.) Thus $\overline{C}_f$ is precisely equal to the weight of the matching. It follows then that an assignment defined by a maximum weight matching for G maximizes $\overline{C}_f$ and thus minimizes C_f, the communication costs incurred by assignment f. As discussed above, an assignment which minimizes C_f is optimal.

As stated above, maximum weight matchings and thus optimal assignments can be found in polynomial time. **Q.E.D.**

Algorithm M:

- Construct a matching in G using a polynomial time algorithm for finding maximum weight matchings.
- Construct an assignment according to the steps described in part (a) of Theorem 1.

Algorithm M is well-suited for the problem of dynamic contraction in ethernet-based distributed systems for regular distributed computations. As discussed earlier, parallelism is maintained in regular distributed computations by contracting an equal number of tasks to each processor. For many distributed algorithms, contraction which assigns two tasks per processor is acceptable and even desirable. In these cases, the number of tasks is precisely equal to twice the number of processors, and Algorithm M can be used to find an optimal contraction by setting $B = 2$. Figure 2 illustrates the contraction of a regular distributed algorithm for the simplex method of linear programming. This algorithm was developed by [6] for execution on the Charlotte Distributed Operating System which consists of 20 Vax 11/750. Other regular algorithms for which contraction to $\frac{k}{2}$ processors is useful include Jacobi iterative method for solving LaPlace equations on a rectangle, successive over-relaxation iterative method for solution of linear systems of equations, Nelson's version of Horowitz and Zorat's matrix multiplcation algorithm. These algorithms all appear in [14].

We also note that many existing distributed systems, such as those in use at academic and research institutions, consist of 40-50 nodes. Algorithm M can thus be used for the optimal assignment of distributed computations consisting of up to twice that many tasks. We claim (but do not substantiate now) that there are a significant number of distributed algorthms that are within these size constraints. In particular, for many distributed algorithms, such as the simplex algorithm, the number of tasks is a user option and can therefore be specified to be in the range necessary for Algorithm M.

3.3. Algorithm H, a Heuristic Algorithm for Task Assignment

Theorem 1 suggests the following heuristic, polynomial-time algorithm for task systems with an arbitrary number of tasks, n identical processors, and bound B, $\lceil \frac{k}{n} \rceil \leq B \leq k$, on the maximum number of tasks per processor.

Algorithm H: This algorithm reduces the original task graph to one containing $\leq 2n$ nodes, each with no more than $\lceil \frac{B}{2} \rceil$ tasks per node. Algorithm M can then be used to produce an optimal assignment for the reduced graph, but this assignment may be suboptimal for the original graph.

- Construct a graph G with a node for each task t_i and an edge between each pair of nodes t_i and t_j with weight c_{ij}.
- If $k \leq 2n$, then Theorem 1 applies and Algorithm M can be invoked to obtain an optimal assignment.
- If $k > 2n$, then tasks are grouped into clusters utilizing the Sort Greedy Algorithm (described below) with a limit $\lceil \frac{B}{2} \rceil$ on the maximum size of a cluster, where $\lceil \frac{k}{n} \rceil \leq B \leq k$. Sort Greedy continues to form clusters until the number of clusters is less than or equal to $2n$.

Symmetric Contraction of A Distributed Algorithm for the Simplex Method *

The system to be solved is represented by a matrix M. and a solution is obtained through repeated iterations on M. At each iteration, it is necessary to (1) select a pivot column from M, (2) select a pivot row from M, and (3) perform operations on each row in M using the values of the elements in the pivot row.

Given m rows in M, we distribute the work to p processes by assigning each **calculator process** m/p contiguous rows. The selection of the pivot column can be done locally by each calculator process. However, the selection of the pivot row involves choosing the "best" row of the m rows in M. One alternative for achieving the distributed voting to select the pivot row utilizes latin squares.

Latin squares voting: Each calculator process has an ordered list of all the other calculator processes such that no two lists contain the same process in the same ordinal position in the list. During each round, a given calculator sends its best row (either its own best, in round 0, or the best seen so far, in later rounds) to the next calculator on its list. During that round, it also receives a message from some other calculator. The lists can be arranged so that after *log p* rounds (base 2), each calculator knows the best row (assuming the number of processes p is a power of 2).

A task graph representing the distributed simplex algorithm designed for 8 tasks is shown below. The communication edges are defined by the latin squares configuration also shown in the figure. Because of the regular nature of the distributed algorithm, $c_{ij} = C$ for all i,j. If contraction of this algorithm is necessary, contraction to 4 processors preserves the parallelism in the algorithm because of the identical nature of the tasks. Thus Algorithm M can be invoked with B=2, k=8, and n=4. to find an optimal assignment with total IPC of 20*C units.

Lists for the Calculator Processes

	1	2	3	4	5	6	7	8
Round 0	2	3	4	5	6	7	8	1
Round 1	3	4	5	6	7	8	1	2
Round 2	5	6	7	8	1	2	3	4

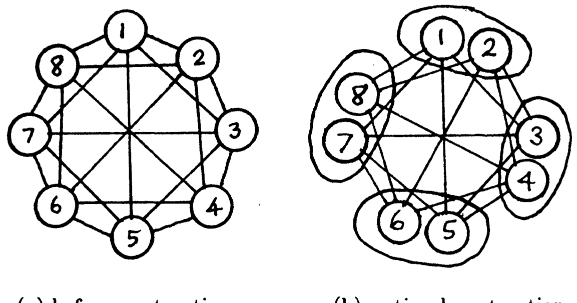


Figure 2: Contraction of a Distributed Algorithm for the Simplex Method

* Only the bare bones of the algorithm are presented here. The full algorithm can be found in [6].

- A new graph G' is constructed with a node corresponding to each cluster and an edge between the pair of clusters r_1 and r_2 with weight

$$w_{12} = \sum_{\substack{t_i \epsilon r_1 \\ t_j \epsilon r_2}} c_{ij}$$

- Since the number of nodes in this new graph G' is less than or equal to $2n$, Algorithm M can be used to produce an assignment of clusters to processors which minimizes the total inter-cluster communication costs while keeping the number of tasks on each processor less than or equal to the bound B.

Subroutine Sort Greedy: This subroutine is a greedy algorithm which groups tasks into clusters of size $\leq \lceil \frac{B}{2} \rceil$ such that the total number of clusters is $\leq 2n$.

```
• Initially, each task in $G$ is in a task group by itself.

• Construct a list $L$ which contains the edges of $G$ sorted in
  non-increasing order.

• While there are unmarked edges remaining

      • • Find the next unmarked edge $e = (t_i, t_j)$ in the
      list $L$. Mark it.
          $G_i$ is the task group containing $t_i$.
          $G_j$ is the task group containing $t_j$.

      • • If $|G_i| + |G_j| \lceil \frac{B}{2} \rceil$ then

              • • • Merge the two groups:
                  $G_{new} = G_i \bigcup G_j$

              • • • Mark all the edges between tasks in
                  $G_i$ and tasks in $G_j$

      • • Else do not merge $G_i$ and $G_j$.
```

Algorithm H is a polynomial time algorithm. The complexity of Subroutine Sort Greedy is $O(e \log e)$ where e is the number of edges in the computation graph. The maximal matching is found using Algorithm M in polynomial time as discussed earlier.

3.4. Simulation Results

In order to evaluate the performance of Algorithm H in finding suboptimal task assignments, simulation runs were performed on a variety of typical task forces. Altogether, 90 task forces were simulated with the number of tasks ranging from 4 to 35 and the number of processors ranging from 2 to 5. Optimal assignments were computed using a branch and bound backtracking algorithm.

The data used in the simulations are organized into four categories. Dataset 1 (*Clustered*) consists of randomly generated task-processor systems in which tasks form clusters. Communication costs between tasks within the same cluster are on the average larger than communication costs between tasks in different clusters. Dataset 2 (*Sparse*) consists of randomly generated task-processor configurations in which the communication matrix is sparse. In particular, the communication costs are nonzero for only $\frac{1}{6}$ of the $\binom{k}{2}$ possible pairs of tasks. Dataset 3 (*Actual*) consists of data representing actual task forces derived from numerical and matrix algorithms, operating systems programs, and general applications programs. In this dataset, specific information about the number of tasks and/or about which pairs of tasks communicate with each other was available in the literature. Estimates of execution and communication costs were made from information such as the number and type of messages passed between tasks, from the function of the tasks, and from raw data on these costs. Dataset 4 (*Structured*) consists of task forces whose task graphs have the structure of a ring, a pipe, a tree, or a lattice. Details about these datasets can be found in [11].

Algorithm H performed extremely well, finding an optimal assignment in 81.1% of cases. Table 1 below shows the ratio $\frac{T_H}{T_O}$ where T_H is the cost of an assignment found by Algorithm H, while T_O is the cost of an optimal assignment.

Table 1: Distribution of Ratio T_H/T_O by Dataset

Dataset	No. of Cases	Percent of Simulations					
		(Optimal) = 1.00	≤ 1.10	≤ 1.20	≤ 1.30	≤ 1.40	≤ 1.50
All Data	90	81.1	91.1	95.5	96.6	98.8	100.0
Clustered	19	47.4	79.0	94.8	100.0	100.0	100.0
Sparse	16	87.5	87.5	87.5	87.5	93.8	100.0
Actual	22	90.9	100.0	100.0	100.0	100.0	100.0
Structured	33	90.9	93.9	96.9	96.9	100.0	100.0

4. Conclusion

This work represents a contribution to the current research effort to utilize the potential of distributed computing systems for parallel computation. To summarize, we have presented two algorithms for the assignment of tasks to processors in order to minimize interprocessor communication costs under the constraint of a bound on the number of tasks assigned to each processor. Algorithm M finds optimal assignments for systems in which the number of tasks is $\leq$ twice the number of processors. Algorithm H is a heuristic for arbitrary task-processor configurations. This model of the task assignment problem and the use of Algorithms M and H are suitable for assignment of tasks and for contraction of tasks in distributed systems which have ethernet as the underlying communication medium.

We are looking at a number of extensions to the work described in this paper. First, because Algorithm H utilizes a greedy type algorithm to reduce the task graph, it is clear that poor assignments may result when the task graph has uniform communication costs. For example, in the binary tree task graphs of Figure 1, an optimal assignment has cost 3 while Algorithm H could yield a poor assignment with cost 8. (Algorithm H could also find the optimal assignment but is not guaranteed to.) Thus, we are interested in refining Algorithm H to handle the case of uniform communication costs. We are also interested in finding algorithms tailored to regular graph structures such as trees, rings, lattices; for these restricted graphs, it may be possible to find optimal algorithms. In the longer run, we are interested multiphase contraction as described in [14] and decentralized dynamic contraction. Many distributed algorithms involve multiple execution phases with a distinct communication pattern associated with each phase. Decentralized dynamic contraction involves local detection of the need for contraction at execution time and achievement of contraction through negotiation among processors rather than through a centralized controller. Both of these problems are related to our interest in process migration in distributed computing systems; in fact, dynamic contraction is essentially carefully-orchestrated process migration. We are currently engaged in a study of parallel and distributed algorithms to determine the role that characteristics of these algorithms can take in guiding process migration in distributed systems.

References

[1] Y. Artsy, H.Y. Chang, and R. Finkel, "Processes Migrate in Charlotte", University of Wisconsin Dept. of Computer Science Technical Report No. 655, August 1986.

[2] F. Berman and L. Snyder, "On Mapping Parallel Algorithms into Parallel Architectures", *Journal of Parallel and Distributed Computing", Vol. 4 No. 5, Oct. 1987, pp. 549-458.*

[3] S.H. Bokhari, "Partitioning Problems in Parallel, Pipelined and Distributed Computing", *IEEE Transactions on Computing,* can't find which issue now, but will find it, 1987.

[4] W. W. Chu, L. J. Holloway, M. T. Lan, and Kemal Efe, "Task Allocation in Distributed Data Processing," *IEEE Computer,* Nov. 1980, pp. 57-69.

[5] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer,* June 1982, pp. 50-56.

[6] R. A. Finkel, "Large-grain Parallelism - Three Case Studies", in *The Characteristics of Parallel Algorithms,* edited by L.H. Jamieson, D.B. Gannon, and R.J. Douglas, MIT Press, 1987, pp. 21-64.

[7] Z. Galil, S. Micali, and H. Gabow, "Priority Queues with Variable Priority and an $O(EV\log V)$ Algorithm for Finding a Maximal Weighted Matching in General Graphs", *23rd Annual Symposium on Foundations of Computer Science,* Nov. 1982, pp. 255-261.

[8] L. Hyafil and R.L. Rivest, "Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems", Report No. 33, IRIA-Laboria, Rocquencourt, France, 1973.

[9] E. Lawler, *Combinatorial Optimzation, Networks and Matroids,* Holt, Rinehart, and Winston, 1976.

[10] V. M. Lo and J. W. S. Liu, "Task Assignment in Distributed Multiprocessor Systems" *Proceedings of the 1981 IEEE International Conference on Parallel Processing,* 1981, pp. 358-360.

[11] V. M. Lo, "Task Assignment in Distributed Systems,", Dept. of Computer Science, University of Illinois, Ph.D. Thesis, October 1983.

[12] V. M. Lo, "Task Assignment to Minimize Completion Time", *IEEE 5th International Conference on Distributed Computing Systems,* May 1985.

[13] V. M. Lo, "Heuristics for Static Task Assignment in Distributed Systems", in press for *IEEE Transactions on Computers.* (Also University of Oregon Technical Report CIS-TR-86-13.)

[14] P. A. Nelson, "Parallel Programming Paradigms", University of Washington Computer Science Technical Report No. 87-07-02, July 1987.

[15] H. S. Stone and S. H. Bokhari, "Control of Distributed Processes," *IEEE Computer,* July 1978, pp. 97-106.

[16] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Engineering,* Vol. SE-3, No. 1, Jan. 1977, pp. 85-93.

The Uniform System: An approach to runtime support for large scale shared memory parallel processors

Robert H. Thomas

BBN Advanced Computers Inc.
10 Fawcett St.
Cambridge MA, 02238

Will Crowther

BBN Laboratories Inc
10 Moulton St
Cambridge MA, 02238

Abstract

Widespread use of large scale shared memory parallel processors requires easy-to-use programming techniques that efficiently exploit parallelism. This paper describes a practical approach to runtime support for parallel programming, called the Uniform System, that is based on the notion of the computational tasks to be performed rather than the processes or processors that perform the tasks. The basis for this approach is a technique for describing many related tasks in terms of a method for generating individual tasks. The paper presents the considerations that shaped the approach, and describes the approach itself in terms of the facilities it provides for parallel programming. It includes several examples illustrating use of the approach, both explicitly by the application programmer and as the target for compiled code that support parallel programming constructs. Finally, the paper describes the Uniform System implementation for the Butterfly Parallel Processor.

1. Introduction

Tightly coupled, shared memory parallel processors with tens to hundreds of processors have become commercially available over the past few years. The ability of machines in this class to provide cost effective conventional time sharing service has been demonstrated, and time sharing can be expected to be an important application for these machines for years to come. However, many of these machines were designed to support computationally intensive applications by exploiting the parallelism inherent in them through parallel processing. Widespread use of these machines to parallel process applications requires techniques for programming parallel implementations that are relatively easy to learn and to use, and that efficiently exploit the parallelism provided by the machine architectures.

Few would argue that a higher level of abstraction than that provided directly by the hardware is desirable. Although the familiar process abstraction, as supported by conventional operating systems, has an important role in runtime support for parallel programming, we believe that by itself, it does not represent an adequate basis for programming applications that use tens to hundreds of processors.

An approach to runtime support for large scale shared memory parallel systems called the Uniform System has been used with success for several years on the Butterfly^{tm} Parallel Processor [BBN87]. The runtime support provided by the Uniform System is used in two ways. Application programs call it directly and compilers generate calls on it. Although at present it is used to support programming language extensions for explicitly controlling parallel activity, it is suitable as the target for automatically detected parallelism.

The Uniform System runs under the two operating systems currently supported for the Butterfly Parallel Processor, Chrysalis [BBN88a] and Mach 1000. Chrysalis is a proprietary real time operating system for the Butterfly Parallel Processor. Mach 1000 is a variant of Mach [Acc86], a new operating system developed at Carnegie-Mellon University that is fully compatible with Berkeley Unix^{tm} version 4.3bsd. It runs on a variety of machines, including several shared memory parallel processors.

There are two common architectural approaches for building shared memory parallel processors. Bus-based architectures connect processors and memory to a high speed bus; switch-based architectures connect processors and memory by means of a high performance interconnecting switch. The fixed capacity of a bus usually limits bus-based architectures to a few tens of processors. Because the capacity of a switch usually scales with the number of processors, switch-based architectures can handle hundreds of processors. Sequent [Seq86], Encore [Enc86] and Alliant [Per86] manufacture bus-based machines. The BBN Butterfly Parallel Processor and the IBM RP3 [Pfi85a] are examples of the switch-based architecture.

There are two commonly used approaches for packaging shared memory systems. One uses separate processor and memory modules, and the other uses combined processor-memory modules. Both bus-based and switch-based architectures accommodate either approach. The different packaging approaches result in two different classes of shared memory machines [You87]:

1. *Uniform Memory Access* (UMA) machines with uniform memory access times for all data.
2. *Non-uniform Memory Access* (NUMA) machines in which a particular processor can access some memory (its local memory) faster than other memory (memory local to other processors).

Although the Uniform System was developed for the Butterfly Parallel Processor, which is a NUMA machine, we believe that the Uniform System approach works well on shared memory machines of both classes, and that the ideas it embodies, if not the interfaces it supports, port relatively easily to other machines

This paper describes the Uniform System approach to runtime support. It presents the considerations that shaped the approach first. Following that, it describes how processors and memory are handled. Next, it gives several examples illustrating use of the Uniform System. Finally, the paper sketches the implementation and discusses some performance considerations. To expedite presentation, the paper simplifies the descriptions of some Uniform System features. Readers interested in the details should consult the programmer's manual [BBN88b].

2. Runtime support for parallel processing

Runtime support for shared memory parallel processors needs to address two principal issues:

1. Processor management. Means must be provided to control how (or at least request how) processors are allocated to an application program throughout the course of its execution.

2. Memory management. Means must be provided to control the access processors have to various regions of memory.

It is useful to introduce some terminology and state some assumptions about the environment within which runtime support exists before describing how the Uniform System approach addresses these issues.

We assume an operating system that supports processes. For our purposes, a *process* is the unit of activity the operating system schedules for execution on a processor. A process is characterized by an address space and a program counter. We assume that the operating system provides means for creating, starting and stopping processes, as well as for establishing sharing relationships among the address spaces of separate processes.

A *task* is a basic unit of computational work. Processes running on processors execute tasks. For the sake of concreteness, it is useful to think of a task as the execution of a subroutine; however, as we shall see shortly, the Uniform System does not restrict tasks in this way.

The Uniform System approach to processor management is based on several beliefs.

Belief #1:

> When there are tens or hundreds of processors it is infeasible to deal individually with each processor and each process.

Belief #2:

> The criteria used by operating systems to schedule processes are typically based on minimal, and at best indirect, knowledge of application program needs. The goal of an operating system is to optimize the utilization of system processor, memory and i/o bandwidth resources across processes with wildly varying requirements.

Belief #3:

> Achieving high application performance by parallel processing requires means for applications to control the assignment of processors to application tasks.

The principal consequences of beliefs #1, #2 and #3 are:

1. From the programming point of view the primary focus is on the computational work or tasks to be done rather than the processes or processors that execute tasks.

2. Although the focus is principally on tasks rather than processes, processes form a fine implementation basis for runtime support. By using one process per processor, the processors can be thought of as computational servers that execute application tasks.

With the view that processes serve as a pool of equivalent workers, the role of the runtime support system is to control the scheduling of tasks to processes. The phrase *process running on a processor* is cumbersome. Since the notion of process is secondary to the notion of task and since the Uniform System implementation uses a single process running on each processor, this paper uses *process* and *processor* interchangeably, being careful to make the distinction whenever important.

Belief #4:

> Since the execution of a single program may involve thousands of tasks, a concise means of specifying many related tasks is preferrable to an explicit enumeration of the tasks.

Belief #5:

> Runtime scheduling of tasks to processors is more flexible than scheduling at compile time or program construction time. Furthermore, runtime scheduling does not preclude more static scheduling schemes.

The consequences of beliefs #4 and #5 are:

1. The basis of the Uniform System processor management mechanism is a technique for describing many related tasks in terms of a method for generating the individual tasks.

2. Processors schedule themselves by using the generation method to obtain their next task. This naturally results in dynamic load balancing behavior with processors looking for work when they complete tasks.

This approach to processor management places requirements on the memory management approach. To treat processors as equivalent workers, the processes running on each must share regions of their address spaces. In particular, application data accessed by tasks must be allocated to shared memory regions to ensure that any processor can work on any task.

The processes need not share their entire address spaces. For example, there is no need to share their call/return stacks or temporary storage used to execute tasks.

The memory management mechanisms provided by the runtime support system should permit programmers to control the *visibility* of data. It should distinguish between *process private* data visible only to a single process and *globally shared* data visible to all processes.

For NUMA machines, like the Butterfly, the *location* of data is often an important performance consideration. Thus, for a NUMA machine, runtime memory management should allow programs to control where (i.e., on which processor-memory module) data is allocated, and provide efficient means for copying blocks of data from one memory to another.

To summarize, the Uniform System treats tasks as distinct from processes in several important ways:

- A single process may execute many tasks (sequentially).
- The operating system specifies the data structure defining a process, the application programmer specifies the data structure of a task.
- Tasks are usually described in sets rather than individually.
- A task will not be started until it is ready to run to completion. There is no virtue in setting aside one task in favor of another.

All processors are treated alike: all can create new task collections via the generator mechanism (described below), all can allocate storage. One processor is special in that it, early on, acquired the task of running the application program. It will be the first processor to invoke the help of others, and the last to terminate on program exit. Otherwise, it is no different from the rest.

3. Initiating and managing parallel activity

Processors running a Uniform System application are in one of 2 states:

- *Busy*, working on an application task.
- *Free*, looking for an application task.

When a processor working on an application task finishes its current task, it looks for the next task. If there are more tasks to be done, it starts working on one them. If not, it enters the free state.

The Uniform System uses shared memory data structures, called *task generators*, to specify work to be done. A task generator data structure describes work in terms of a related collection of tasks. For example, a task generator can represent the work to be done by a loop; each loop iteration is a task, and the set of all loop iteration tasks represents all of the loop work.

In general, a task is characterized by:

- A body of code, called the *task work procedure*, that is executed to perform the task.
- Some task-specific parameters.
- Data common to all of the tasks associated with a given task generator.

A task generator specifies how to obtain the next task in the collection of tasks associated with it. As such, it is an implicit description of work rather than an explicit enumeration of all tasks in the work collection. This is important when the enumeration might include many thousands of tasks.

A task generator data structure includes:

- A *task generation procedure*. This is a body of code executed to obtain the next task associated with the task generator. When a process starts working on a task generator it calls the task generation procedure.

  The task generation procedure typically loops generating tasks and executing them until all tasks associated with the generator have been generated. An individual task is described in terms of the parameters for a task work procedure. Task generation involves generating the parameters for the next task, and task execution is accomplished by executing the task work procedure with those parameters.

  The task generation procedure for the loop example above is particularly simple. It is a loop that atomically increments a counter to produce an index. If the index is within range, it identifies the next loop iteration to execute. If it is not within range, work on this task generator can cease since all loop iterations have been scheduled. In this case the task generation procedure exits.

- A *task work procedure*. This is a body of code executed to perform a task. The parameters obtained by executing the task generation procedure are used with the task work procedure.

  For the parallel loop example, the task work procedure is the loop body, and its parameter is the index generated by the task generation procedure that specifies a particular loop interation.

- Data. This specifies data that is accessible to all of the tasks associated with the task generator.
- Fields for bookkeeping. The task generation procedure uses these for keeping track of its progress.

  For the parallel loop example, a counter and a range suffice for bookkeeping.

When a program starts to run on a group of processors, its execution is limited to a single processor until it initiates parallel activity by placing a task generator data structure into shared memory. Free processors find the task generator and immediately use it to generate and work on its tasks. Processors currently busy will begin working on the new task generator when they finish their current work.

The Uniform System runtime support library provides a collection of routines, called *activators*, for initiating parallel activity. These routines create and intialize task generator data structures and place them in shared memory where processors can find them. There are a variety of *specific activators* that allow users to specify some task generator fields and default others. In addition, there is a *general purpose activator*, called *ActivateGen*, that provides precise control over the task generator created. All of the specific activator routines call *ActivateGen.*

Perhaps the simplest specific activator is the index activator:

```
GenOnI (task_work, range)
```

This activator specifies a task work procedure and a range parameter. The task generation procedure is implicit. It loops, atomically incrementing a counter to generate the tasks:

```
task_work(0), task_work(1), ...
        task_work(range-1)
```

Any data common to all of the tasks is implicit, and assumed to be in shared memory.

GenOnI can be used to parallelize loops. For example, consider the serial C language loop:

```
    ...
    for (i = 0; i < N; i++)
        a [i] = b [i] * c [i];
        ...
```

This loop has no data dependences and can be transformed into the parallel loop:

```
    ...
    simple_loop_body (i)
        int i;
    {   a [i] = b [i] + c [i];
    }
    ...
    GenOnI (simple_loop_body, N);
    ...
```

Other simple specific activators include the array activator:

```
    GenOnA (task_work, range1, range2)
```

a 2 dimensional version of *GenOnI* that generates tasks of the form:

```
    task_work(row, column)
```

corresponding to each element in a *range1* x *range2* array, and the half array activator:

```
    GenOnHA (task_work, range1, range2)
```

a variant on *GenOnA* that generates tasks of the same form corresponding to elements of the lower triangular portion of a *range1* x *range2* array.

The activator

```
    GenTaskForEachProc (task_work, task_data)
```

where *task_data* points to task data in shared memory generates exactly one task for each processor of the form:

```
    task_work(task_data)
```

These simple activators support a style of parallel programming, sometimes referred to as single program multiple data (SPMD) programming [Kar87], where each processor executes the same code against different data. They are also capable of supporting less homogeneous situations. For example, the index passed to *GenOnI* tasks can be used to dispatch to index-specific code. The activator

```
    GenTasksFromList (routine_list, arg_list)
```

generates a set of tasks, one for each routine-argument pair from the routine and argument lists. Example 2 in Section 5 illustrates further how the Uniform System generator mechanism can be used to support heterogeneous situations.

The simple activators described above are sufficient for many applications. However, there are situations where it is desirable to exercise more control over generators.

Two activator control disciplines are supported:

1. *Synchronous activators* return to the caller after all of the generated tasks have been executed. Furthermore, the processor that calls a synchronous activator always works on tasks that result.

   In simple situations where only one processor calls synchronous activators, each call represents a barrier synchronization; execution will not procede beyond the call until all of the tasks associated with it have been completed.

2. *Asynchronous activators* return to the caller as soon as the generator has been activated. This enables the caller to do other work either directly or by invoking other activators. The caller can later work on the generated tasks if it so chooses.

Normally, when a generator is active, processors begin working on the generator until either all processors are working on it, or all the tasks have been generated. In situations where multiple generators can be active simultaneously, it may be desirable to control the number of processors used for each. Means exist to limit the processors used on a generator. For example, the code fragment:

```
    Asynch_limited_gen_1 (task_work_1,
                          processor_limit_1, ...);
    Asynch_limited_gen_2 (task_work_2,
                          processor_limit_2, ...);
```

activates two generators, each limited to a subset of the total available processors, in a way that enables the activating processor to work on other things.

In some situations it may be possible to place an upper bound on the number of tasks required, but the actual number required might be data dependent and significantly less than the bound. For example, consider a search that can be partitioned into N tasks, each of which searches one of N disjoint regions of the search space. If a task for one of the first regions succeeds, it is unnecessary to generate tasks to search the remaining regions. Means exist to abort task generation in such situations.

In some situations when a processor begins working on a generator, it is convenient for it to execute some generator-specific initialization code prior to executing any of the generator's tasks. For example, it might be convenient to initialize some private temporaries required by the tasks. Similarly, it is sometimes convenient for a processor to execute epilog code after it has executed the last task for a generator; for example, to combine its results with those of other processors working on the generator. Initialization and finalization procedures can be specified when a generator is activated.

All the specific activators in the Uniform System library call *ActivateGen*, the general purpose activator. *ActivateGen* can also be used directly to build custom activators when none of the specific activators is well matched to an application's needs. The parameters for *ActivateGen* specify:

- A task generation procedure
- A task work procedure
- A pointer to data common to all of the tasks
- Generator initialization and finalization procedures

- A flag indicating whether the generator is synchronous or asynchronous
- A flag indicating whether the generator is abortable
- A processor limitation.

When programs call Uniform System activators directly, they specify subroutines for the task work and task generation procedures. The task generation subroutine usually loops generating parameters it uses to call the task work subroutine. When a compiler generates calls on the Uniform System to support parallel constructs, it can compile a single procedure that embodies both the task generation procedure and the task work procedure. The resulting procedure is, in effect, a task generation procedure that includes in-line expansion of the task work procedure.

Finally, programs can nest calls to activators. When a processor (P) begins working on a generator (G1) it continues working on it until all of its tasks have been generated or a task it is working on calls an activator. When this occurs, P, as well as any free processors, begin working on the new generator (G2). When generator G2 is finished, P resumes working on generator G1.

4. Managing memory

The Uniform System provides two types of memory regions for application program data: process private memory and globally shared memory.

Process private memory is accessible only to the allocating process. The Uniform System uses conventional uniprocessor memory allocation mechanisms, such as the Unix *malloc* family of dynamic memory allocators, to support process private memory

Globally shared memory is visible to all of the processors. To support it, the Uniform System uses mechanisms that allocate storage in regions of the process address space that are shared among all processes. The basic Uniform System allocator for globally shared memory is:

```
UsAlloc (number_of_bytes)
```

which is analogous to the *malloc* allocator.

Dynamic storage allocation is foreign to Fortran77. To permit Fortran programs to explicitly manage globally shared memory, Butterfly Fortran has been extended to support dynamic storage allocation along the lines proposed for Fortran8x. In addition, it has been extended to support a *visibility* attribute for common blocks that allows programmers to declare whether a common block is to be process private (the current default) or globally shared:

```
shared common /dat1/ real a(1000), real b(20)
```

The Fortran compiler generates calls on the runtime support provided by the Uniform System to implement globally shared common.

For NUMA machines, like the Butterfly, the location of data relative to the processor that accesses it can have a significant effect on program performance. The Uniform System provides means to control where storage is allocated and means to move blocks of data efficiently from one memory to another.

UsAlloc allocates storage on the local processor if there is space; if not, it cycles through other processors until it finds one with enough free storage to satisfy the request. In addition,

```
UsAllocLocal (number_of_bytes)
```

allocates from the memory of the local processor, and

```
UsAllocOnProc (processor, number_of_bytes)
```

allocates storage from memory of the processor specified.

When space for a data structure must be allocated, it may not be possible to tell which processor will access it most heavily. In such situations, a block transfer routine can be used to move the data to the processor accessing it (and back again, if processing changes it). The block transfer trades off the cost of moving data against faster data references when the data is local, and is used when there are sufficient data references to recover the cost of the transfer.

On a NUMA machine, if a processor allocates all of its data to a single memory module, parallel execution will be serialized when multiple processors simultaneously access the memory module. Clearly, the application data should be scattered across the available memories so that the full memory bandwidth of the machine can be used to remove this bottleneck. Some machines interleave consecutive memory addresses across memory to achieve this sort of scattering. The Butterfly hardware does not interleave addresses; within a page, consecutive memory addresses fall within the same processor-memory module. As a result software must scatter the data.

To facilitate data scatterering, the Uniform System provides the scattering memory allocator:

```
UsAllocScatterMatrix (row, columns,
                      element_size)
```

which allocates space for a 2 dimensional array by allocating successive rows on different memory modules and storing pointers to the rows in a vector. Because the C language permits the same syntax for array subscripting and pointer dereferencing, a program can reference elements of a scattered array like a standard C array. Of course, for a scatter array to effectively reduce memory contention, each processor accessing it should have its own local copy of row pointers;. Otherwise, the memory holding the vector of pointers will serialize access to it.

Butterfly Fortran has been extended to support a scatter attribute for arrays (which are scattered by column rather than row) and for common blocks. The declaration:

```
shared scattered common /dat1/
        real a(1000),real b(20)
```

causes the common block data to be scattered across available memories.

5. Examples

This section contains three examples that illustrate use of the Uniform System. The first example illustrates direct calls upon the Uniform System and discusses some performance issues. The second illustrates compiling into Uniform System runtime support, and the third illustrates the construction of a custom activator. Space limitations do not permit the examples to include complete programs; instead, program fragments abstracted from working Butterfly programs are used.

Example #1. Direct use of the Uniform System.

This example computes the product (*a*) of two matrices (*b* and *c*) by computing individual elements of the product matrix in parallel. The example uses an array activator that generates a task for each element in *a*. The task work procedure is a routine that computes the dot product of a row of *b* and a column of *c*, and stores the result in *a*. A C language code fragment that implements this is:

```
typedef struct
{   int size;
    float * * a, * * b, * * c;
} matrix_problem;

dot_product (p, row, col)
    matrix_problem * p;
    int row, col;
{   int i;
    float temp = 0.0;

    for (i = 0; i < p->size; i++)
        temp += p->b[row][i] * p->c[i][col];
    p->a[row][col] = temp;
}

matrix_multiply (a, b, c, size)
    float * * a, * * b, * * c;
    int size;
{   matrix_problem * p =
      (matrix_problem *)
        UsAlloc (sizeof (matrix_problem));

    p->size = size;
    p->a = a;
    p->b = b;
    p->c = c;
    GenOnAFull (dot_product, p,
                p->size, p->size, ... );
}
```

Matrix_multiply accepts the operand and result matrices (assumed to be in shared memory) and creates a structure to hold data common to the dot product tasks. It then uses *GenOnAFull*, a variant of the array activator, to start the parallel computation; this variant accepts a pointer (*p*) to the common data and causes tasks of the form:

```
dot_product (p, row, col)
```

to be generated. *Dot_product* stores the specified dot product in the result matrix *p->a[row][col]*.

This program must be tuned to achieve best performance on a large scale NUMA machine like the Butterfly:

1. The matrices would be stored as scatter matrices to reduce memory contention.
2. *Dot_product* accesses an entire row and an entire column of the operand matrices. Since elements in a row of a scatter matrix are contiguously stored, *dot_product* would use the block transfer operation to make a local copy of the row. Elements in the columns are scattered, however, preventing use of block transfer in this way. Transposing a matrix is inexpensive relative to mutiplying two matrices, and can be performed very efficiently in parallel. An optimized version would transpose the *c* matrix before activating the array generator. This would allow *dot_product* tasks to block transfer the columns of *c*.
3. With optimization #2 each dot product requires 2 block transfer operations. Note that each row of *b* and column of *c* is actually used for N dot products. By clumping tasks such that each computes an n x n rectangle of dot products, n^2 dot products can be computed using only 2•n block transfers.

   Clumping reduces the block transfers required. It also reduces the number of tasks generated. This further improves performance, up to a point, by reducing the overall task generation overhead required by the program. The benefit of clumping diminishes when there are too few tasks to prevent processor starvation effects that occur near the end of the computation from dominating the runtime.

Optimizations #1 and #2 would not be beneficial on a UMA machine. Clumping would improve performance by reducing overall task generation overhead, however. Furthermore, for a UMA machine with a data cache, it would increase the likelihood that needed row and column elements were resident in the cache.

Example #2. Compilation into the Uniform System.

Butterfly Fortran extends Fortran77 with directives for explicitly specifying parallel activity. This example outlines how the compiler uses Uniform System support to implement a *DO PARALLEL* directive for parallel loop execution. (Butterfly Fortran enables interloop data dependences to be properly handled, but this issue is not addressed here.)

DO PARALLEL can be used to transform the serial loop

```
integer i, k, a(1000)

do i = 1, 1000
    k = a (i)
    a (i) = x (k * k)
end do
```

into a parallel loop:

```
do parallel, shared (a), private (k)
do i = 1, 1000
    k = a (i)
    a (i) = x (k * k)
end do
```

The *shared* and *private* options with *DO PARALLEL* specify that *a* is to be accessed by all processors, and *k* is to be private to each.

The compiler transforms this parallel loop into the following (pseudo code):

```
integer i, k, a(1000)
record arg_vec
allocatable arg_vec
external Proc123

arg_vec = UsAlloc (48)
arg_vec.a = a
arg_vec.k = k
arg_vec.i = 1

call ActivateGen (..., Proc123, arg_vec, ...)

a = arg_vec.a

...

subroutine Proc123 (arg_vec)

integer i, k
record arg_vec

k = arg_vec.k
a: loop
    i = atomic_add (arg_vec.i, 1)
    if (i > 1000) exit a
    k = arg_vec.a (i)
    arg_vec.a (i) = x (k * k)
end loop

end
```

The general purpose activator, *ActivateGen*, initiates parallel execution of the loop. The compiler constructs the procedure *Proc123* containing the loop to act as the task generation procedure. It uses the structure *arg_vec* to pass processors data they need to execute loop iterations. After initializing the processor private variable *k*, *Proc123* loops, atomically incrementing the globally shared loop counter producing indices used to execute loop iteration tasks in-line.

An option to *DO PARALLEL* controls how loop iterations are scheduled. The loop in this example is self scheduling in that processors working on the loop schedule the iterations. Less dynamic scheduling can be achieved by clumping iterations; e.g., by incrementing the shared loop counter (*arg_vec.i*) by the desired clump size. Fully static scheduling over a fixed number of processors can be achieved by compiling a task generation procedure that simply assigns each processor its share of the iterations as a single task.

This example and the previous one illustrate the SPMD method of parallel programming, where each processor executes the same code (e.g., *dot_product* in Example 1, loop body in this example) against different parts of the data. This is sometimes also called *data partitioning* [Ost86]. The Uniform System also supports a second method, called *functional partitioning*, where each processor executes different code on shared data.

Butterfly Fortran supports a *BEGIN PARALLEL* construct for functional partitioning:

```
begin parallel
    <branch 1>
next parallel
    <branch 2>
next parallel
    ...
end parallel
```

where each branch is a group of statements executed in parallel. Since activators can be nested, there can be parallelism within each branch as well as across branches. To translate *BEGIN PARALLEL* the compiler builds a task generation procedure with a loop that contains a case statement corresponding to each branch. Processors generate tasks by atomically incrementing a shared loop counter that specifies a branch to execute. The semantics of *BEGIN PARALLEL* are similar to the *PCASE* construct of the FORCE package [Jor86].

Example #3. A custom task generator.

Custom activators provide a way to extend the Uniform System. The previous example illustrated how a compiler can construct simple custom activators to support parallel loops. This example outlines the construction of a more complex generator appropriate for applications that use tree-walking techniques.

Many problems contain substantial computational phases that involve searching for a path through a tree to a leaf node that satisfies some solution criteria. Typically, the tree is constructed as the search proceeds. Sometimes a single solution is sought, sometimes all solutions are sought, and sometimes a node close enough by some criteria to an exact solution is sufficient.

For example, the objective of the n-queens puzzle is to place n queens on an n x n chess board so that no queen can attack another. A search for a solution can be organized by constructing a tree whose nodes correspond to legal board configurations. The root node corresponds to the empty board, and children nodes correspond to various legal queen placements. In the tree for the 4-queens problem the root node would have 4 descendent nodes, each corresponding to a board with a queen in one of the columns of the first row. Each of these depth 1 nodes would have descendent nodes corresponding to legal placements of the second queen in the second row, and so forth. All depth 4 nodes represent boards holding 4 queens that solve the puzzle.

It is possible to formulate parallel solutions to combinatorial problems in terms of tasks that construct and explore the corresponding problem description tree. Two routines can be used to express these solutions. The first adds a new node to an existing node in the description tree:

```
AddNode (existing_node, new_node_data)
```

New_node_data is problem-specific data to be embedded in the new node. For the n-queens problem, *new_node_data* would be the board configuration corresponding to a node. The second routine is a activator that generates tasks corresponding to tree nodes:

```
GenTreeTasks (init, process_node, final,
              root_node_data)
```

where *process_node* is the task work procedure, *root_node_data* is problem-specific data to be embedded in the root node of the tree, and *init* and *final* are initialization and finalization routines. It generates tasks of the form:

```
process_node (node)
```

that explore a node, possibly using *AddNode* to create other tree nodes to be explored by other tasks.

The following C language program fragment finds all solutions to the n-queens puzzle in parallel:

```
init_queens ()
{   my_ solns = Allocate space for this
                     processor's solutions;
}

place_queens (board)
{   next_row = # pieces on board;
    for each column
    {  if (OK to place queen at
               (next_row,  column))
       {   new_board = board +
                        (next_row, column);
           if (new_row == last_row)
             my_ solns = my_ solns +
                               new_board;
           else
             AddNode (board, new_board);
       }
    }
}

final_queens ()
{ total_ solns [Atomic_add (soln_index, 1)]
                  = my_ solns;
}

main ()
{   soln_index = Allocate space in shared
               memory for counter;
    total_ solns = Allocate space in shared
               memory to hold pointers to
               solutions from processors;
    GenTreeTasks (init_queens, place_queens,
               final_queens, empty_board)
}
```

Each processor maintains a list of solutions (*my_solns*) it finds. Before working on any node tasks, each calls *init_queens* to initialize its individual solution list. The *place_queens* task work procedure accepts a legal board configuration with queens in its first k rows, and creates a new board configuration for each column in the next row where it is legal to place a queen. If the next row is the last row, the new board configuration is a solution and the procesor adds it to its solution list. If not, it creates a new tree node corresponding to the new board configuration. When each processor finishes, it calls *final_queens* to combine its solutions with those found by other processors.

For this to work, the task generation procedure used by *GenTreeTasks* must explore the entire problem tree. There are numerous tree walking strategies. Depth first is a particularly simple one. A purely depth first walk is inherently serial, however, since only one processor can follow the depth first path through the tree. The task generation procedure outlined below approximates depth first order: one processor (not necessarily the same one) follows depth first order, and others are pulled along behind it, exploring tree nodes above and to the right of the depth first processor.

The task generation procedure uses the bookkeeping fields in the task generator data structure to keep track of two nodes within the evolving problem tree. One, the *depth first node*, is the node currently being explored by the processor following depth first order. The second is an *allocation node*, used to assign nodes to processors not following depth first order.

When the task generation procedure needs to assign a new node for the processor following depth first order there are two cases:

- If the current depth first node is not a terminal node, it advances the allocation node to the depth first node. Advancing the allocation node in this way forces other processors to follow along behind the depth first processor (see below). It then advances the depth first node to its left most descendent, and assigns the new depth first node to the processor.

- If the current depth first node is a terminal node, the task generation procedure advances in depth first order from it, ignoring any nodes that have already been explored. If it encounters an unexplored node, it advances the depth first node to the node, and assigns it to the processor. If it enounters a node currently being explored, it advances the depth first node to the node, and then uses the allocation node to find the next task for its processor (see below). If it reaches the root without encountering an unexplored node or one currently being explored, the tree has been fully explored, and it exits, freeing its processor for other work.

When a processor not working on the depth first node finishes with a node, the task generation procedure uses the allocation node to find the next node to assign it. (Note that the node a processor is working on can be made the depth first node by another processor.) If the allocation node has unexplored children, the task generation procedure assigns the processor the left-most unexplored child. If not, it advances from the allocation node in depth first order looking for an unexplored node. If it encounters one, it assigns the node to the processor. If not, it waits for some other processor to advance the allocation node or for the depth first processor to exit. If the allocation node advances, the processor repeats this procedure; otherwise it exits.

6. Implementation

The Uniform System runs on the Butterfly Parallel Processor, which is a NUMA machine. It runs under both the Chrysalis and Mach 1000 operating systems .

Chrysalis is a multi-user operating system which allocates groups of processors, called clusters, to users. A Uniform System program running under Chrysalis uses the processors within a processor cluster. As currently supported on the Butterfly machine, Mach 1000 is a multi-user system that manages two processor partitions. There is a public partition shared among all logged in users, and a dedicated partition, from which processor clusters can be allocated on a dedicated basis. Under Mach 1000, a Uniform System program can run either within the public processor partition, in which case it shares processor and memory resources with other time sharing users, or it can run within a processor cluster from the dedicated processor partition, in which case it has exclusive access to the processor and memory resources within its cluster.

A substantial part of the Butterfly implementation deals with the NUMA nature of the machine. However, several key characteristics of the implementation would remain in UMA implementations:

1. Dependence upon efficient synchronization mechanisms. The Butterfly hardware implements efficient atomic *fetch*

and op operations. These are similar in function to the Ultracomputer *fetch and op* operations [Got83] but are implemented entirely at the memory, rather than supported by combining within the switch.

2. Dependence upon operating system support for establishing sharing relations among process address spaces.

3. Use of a single process per processor as a computational server for its processor to avoid unnecessary process context switches.

The following sketches the task generator and storage management implementations for the Butterfly Parallel Processor.

The task generator mechanism must solve two problems:

1. Notification problem. A low latency scheme is required to notify free processors when a new generator has been activated.

2. Bookkeeping problem. An efficient, race-free, deadlock-free scheme is required to keep track of multiple and nested generators.

We outline the approach taken to the notification problem in the Butterfly implementation. Space limitations prevent discussion of the bookkeeping problem.

Although both Butterfly operating systems support interprocess communication (IPC), we rejected solutions based on IPC support as too slow. The basis for a simple low latency approach is a flag in shared memory that indicates whether there is an active generator. Free processors looking for work would spin on the flag. To activate a generator, a processor would store the task generator in a pre-agreed upon place in shared memory and set the flag, which would release any spinning processors.

As described, this might work well on UMA machines with a few processors and clever caching schemes. It does not work well on NUMA machines with hundreds of processors, such as a large Butterfly system, however. The problem is that the memory holding the flag becomes very hot [Pfi85b], [Tho86]. This has two undesirable consequences: there is a severe serial bottleneck in accessing the flag, resulting in a high average latency for generator startup; and, working processors that access data in the memory holding the flag are slowed.

The attraction of this approach is that, at least in principle, the spin frequency can be adjusted to achieve the desired latency. The problem with it is that spinning processors interfere with one another and with working processors. A refinement is to have each free processor spin on a flag in a shared portion of its own memory. This prevents spinning processors from interfering with one another and limits the impact their spinning can have on working processors. At initialization time, every processor "publishes" its spin location, and whenever a processor activates a generator, it writes into each spin location. For a system with a few processors this approach works well, but when there are hundreds of processors, writing into each spin location becomes a serial bottleneck.

A further refinement is to remove this serial bottleneck by making the notification be logarithmic. The processor activating the generator sets the spin locations of a few processors, and when each notices its location has changed, each sets the spin locations of a few more processors, and so forth, until all have been started. For this to actually work, a few details must be addressed. For example, the logarithmic startup tree must be limited to free processors, since a working processor at an intermediate level in the tree would not notifiy processors at lower levels until it becomes free itself.

The implementation uses operating system support for shared memory to set up a large address space region that the processes on each processor share. Because the Butterfly is a NUMA machine, the implementation partitions the shared region into subregions corresponding to processors, and uses the physical memory of each processor to support the corresponding subregion.

The Uniform System shared memory allocation mechanism for the Butterfly machine uses a first fit storage allocation scheme [Knu68] adapted for a NUMA environment. It maintains a free list for each subregion of the address space, and protects each subregion free list with a mutual exclusion lock. The locks support a locking discipline that permits simultaneous allocation activity within different subregions, but serializes allocation within a single subregion.

An implementation for a UMA machine would not partition the shared region of the address space into subregions. For a UMA machine it may be desirable to permit simultaneous allocation activity, however. This would require a more sophisticated locking scheme than can be supported using a single mutual exclusion lock for the allocator [El187].

7. Conclusions.

The Uniform System is a practical approach to runtime support for shared memory parallel processors. It manages parallel activity by means of a task generation mechanism that provides a concise and efficient way to represent large numbers of related tasks. The Uniform System library provides a collection of task generators that are well matched to many applications, and the ability to construct custom generators provides a natural extension path.

The approach was shaped by the beliefs outlined in Section 2. Not all would subscribe to these [LeB86]. Other approaches to runtime support, such as the Force [Jor86] and the monitors package developed at Argonne National Laboratory [Lus83], appear to be motivated by similar views, however. The Uniform System generator mechanism represents an efficient generalization of the parallel processing features these approaches provide.

The Uniform System has been in use for over 3 years running under 2 operating systems on 2 generations of the Butterfly hardware at over 70 user sites. Experience with it over this time has shown that it represents a viable approach to runtime support for large scale shared memory parallel processors. The Uniform System is easy to learn and to use, and it has been used across a wide range of problem domains to produce high performance programs for large scale parallel machines [Cro85], [Kim87], [Gil86], [ONe87], [Jef88].

References

[Acc86] M.J. Accetta et al., "MACH: A New Kernel Foundation for Unix Development", Proc Summer Usenix, July, 1986.

[BBN87] BBN Advanced Computers, "Butterfly Product Overview", 1987.

[BBN88a] BBN Advanced Computers, "Butterfly Parallel Processor Chrysalis Programmer's Manual", 1988.

BBN88b] BBN Advanced Computers, "Programming with the Uniform System", 1988.

[Cro85] W. Crowther et al., "Performance Measurements on a 128-node Butterfly Parallel Processor", Proc 1985 International Conference on Parallel Processing, pp531-540, IEEE Computer Society Press, 1985.

[Ell87] C.S. Ellis and T.J. Olson, "Parallel First Fit Memory Allocation", Proc 1987 International Conference on Parallel Processing, pp502-511, IEEE Computer Society Press, 1987.

[Enc86] Encore Computing Corporation, "UMAX 4.2 Programmer's Reference Manual", 1986.

[Gil86] J. Gilmer, G. Hartwig, and L. Kokinakis, "Parallel Entity Centerered Simulation on the Butterfly Computer", Proc 1986 International Conference on Parallel Processing, IEEE Computer Society Press, 1986.

[Got83] A. Gottlieb et al., "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer", IEEE Trans on Computers, C-32, 1983.

[Jef88] W. Jeffrey et al., "Functional Optimization and Pattern Selection in Rayleigh-Benard Convection: An Implementation on the BBN Butterfly Parallel Processor", submitted to the Journal of Computational Physics.

[Jor86] H. Jordan, "Structuring Parallel Algorithms in an MIMD, Shared Memory Environment", Parallel Computing, 3 (1986), North-Holland, 1986.

[Kar87] A. Karp, "Programming for Parallelism", IEEE Computer, May 1987, pp 43-57.

[Kim87] O. Kimball, L. Cosell, R. Schwartz, and M. Krasner, " Efficient Implementation of Continuous Speech Recognition on a Large Scale Parallel Processor", Proc 1987 International Conference on Acoustics, Speech and Signal Processing, Dallas, TX, 1987.

[Knu68] D. Knuth, "The Art of Computer Programming: Volume 1 Fundamental Algorithms", p435-437, Addison-Wesley, 1968.

[LeB86] T.J. LeBlanc, "Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocess: A Case Study", Proc 1986 International Conference on Parallel Processing, pp764-771, IEEE Computer Society Press, 1985

[Lus83] E.L Lusk and R. A. Overbeek, "Implementation of Monitors: A Programming Aide for the HEP and Other Parallel Processors", Report No. ANL-83-97, Argonne National Laboratory, 1983.

[ONe87] E. O'Neill, E. Tenenbaum, H. Allik, and S. Moore, "Finite Element Analysis on the BBN Butterfly Multiprocessor", Proc 2nd International Conference on Supercomputing, 1987.

[Ost86] A. Osterhaug, "Guide to Parallel Programming", Sequent Computer Systems, 1986.

[Per86] R. Perron and C. Mundie, "The Architecture of the Alliant FX/8 Computer", Digest of Papers, Compcon, Spring 1986. A.G. Bell, ed., IEEE Computer Society Press, 1986.

[Pfi85a] G.F. Pfister et al., "The IBM Research Parallel Processor Prototype (RP3)", Proc 1985 International Conference on Parallel Processing, pp764-771, IEEE Computer Society Press, 1985.

[Pfi85b] G.F. Pfister and A. Norton, "Hot Spot Contention and Combining in Multistate Interconnection Networks", Proc 1985 International Conference on Parallel Processing, pp790-797, IEEE Computer Society Press, 1985.

[Seq86] Sequent Computer Systems, Inc., Dynix Programmer's Manual, 1986.

[Tho86] R. Thomas, "Performance of the Butterfly Parallel Processor in the Presence of Memory Hot Spots", Proc 1986 International Conference on Parallel Processing, pp46-50, IEEE Computer Society Press, 1986.

[You87] M. Young et al., "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System.", Proc. 11th ACM Symposium on Operating Systems Principles, November 1987, pp 63-76.

Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System

Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh

University of Rochester
Department of Computer Science
Rochester, NY 14627

ABSTRACT

The Psyche project at the University of Rochester aims to develop a high-performance operating system to support a wide variety of models for parallel programming. It is predicated on the conviction that no one model of process state or style of communication will prove appropriate for all applications, but that shared-memory multiprocessors (particularly the scalable "NUMA" variety) can and should support all models. Conventional approaches, such as shared memory or message passing, can be regarded as points on a continuum that reflects the degree of sharing between processes. Psyche facilitates dynamic sharing by providing a user interface based on passive data abstractions in a uniform virtual address space. It ensures that users pay for protection only when it is required by permitting lazy evaluation of protection policies implemented with keys and access lists. The data abstractions define conventions for sharing the uniform address space; the tradeoff between protection and performance determines the degree to which those conventions are enforced. In the absence of protection boundaries, access to a shared abstraction can be as efficient as a procedure call or a pointer dereference.

Introduction

Though shared-memory multiprocessors have existed for over 20 years, the design of operating systems for such machines has seldom been the subject of research. For one thing, individual processors have tended to be very few in number, or less than general-purpose. With the notable exception of projects at CMU [26, 33], it is only in recent years that multiprocessors have been constructed with relatively large numbers of equally powerful nodes. It is understandable, then, that the parallel operating systems community has for the past decade focused its attention on loosely-coupled systems, in which more-or-less conventional processors exchange messages over a local-area network.

With the advent of large-scale commercial multiprocessors, several vendors have adapted the UNIX operating system for use on parallel machines. Most message-based operating systems can be implemented on shared-memory machines as well. The Mach project [1] at CMU represents, to a large extent, the merger of Berkeley UNIX with the Accent network operating system [30]. Mach now runs on several multiprocessors, including DEC, Encore, and Sequent machines.

Our aim in the Psyche project is to develop a programming environment (starting with an operating system) that supports truly general-purpose parallel computing. By this we mean that the operating system will run almost any application for which the hardware is appropriate, and will usually run it well. As with the parallel UNIX designs, we also mean that Psyche will not be a back-end system. In addition to individual, highly-parallel applications, it will support large numbers of users with smaller applications, in the style of conventional time-sharing.

We see at least two dangers in adapting an existing operating system for use on a multiprocessor. First, it may fail to provide abstractions that are appropriate for certain applications. Second, it may fail to make effective use of the hardware. Through the course of considerable experience with application and system software, we have become convinced that no one model of process state or style of communication will be best for all parallel applications. Just as a general-purpose operating system for a uniprocessor must support a wide variety of models (e.g. programming languages) for sequential computing, so too must a general-purpose operating system for a multiprocessor support a wide variety of models for parallel computing. Since parallel computing involves concepts (such as scheduling and interprocess communication) that have traditionally been the province of operating systems, parallel versions of traditional operating systems are unlikely to provide the flexibility required by users.

Our first goal for Psyche is therefore *flexibility*: users should be able to implement a wide variety of models for interprocess communication and lightweight process structure. Pieces of an application written under different models should be able to interact easily, that is, to arrange dynamically to share access to arbitrary abstractions. Since Psyche is to be a multiuser system, our second goal is *protection*: it should be possible to associate a protocol with a shared abstraction in such a way that access to the abstraction is possible only by executing the protocol. Finally, since multiprocessors are attractive primarily for speed, our third goal is *performance*: the cost of a simple operation on a shared abstraction should be much closer to that of a procedure call than to that of sending a message in current network operating systems.

Though protection and performance are conventional goals, our emphasis on flexibility is distinctive and unusual. In order to permit user-level control over processes and communication, we have adopted a kernel/user interface consisting of unusually low-level primitives. We do not expect this interface to be easy to use, but the assumption is that most programmers will never attempt to use it. Instead, they will rely on pre-existing libraries and language support packages for process management and communication. We have adopted the position that an operating system kernel should provide only the lowest common denominator for things that will be built upon it. The purpose of the kernel is to provide protection and to hide the most unpleasant idiosyncrasies of the hardware while leaving the bulk of its power available to the language and library builder.

This conception of the role of the operating system does not appear to have guided most recent research projects.

This work was supported in part by NSF CER grant number DCR-8320136, DARPA ETL contract number DACA76-85-C-0001, and an IBM Faculty Development Award. We thank the Xerox Corporation University Grants Program for providing equipment used in the preparation of this paper.

Message-based operating systems, such as Eden [3], Mach [1], and V [14], have tended to provide a kernel interface that is too low-level to be used directly (witness the proliferation of remote procedure call stub generators), yet too high-level to permit alternative approaches to naming, buffering, error recovery, or flow control (we argue this point in [31]). Similarly, most implementations of parallel programming languages have either employed a special-purpose kernel (as in SR [4], StarMod [21], or Linda [13]), or have been built on top of an existing uniprocessor operating system, most often UNIX. We are unaware of any work specifically addressing the design of a kernel to support multiple programming models.

Motivation

Shared Memory Versus Messages

Conventional wisdom holds that parallel processes must communicate either by sharing memory or by exchanging messages. These alternatives are generally viewed as incompatible opposites. It is our contention, however, that conventional approaches are better regarded as points on a *continuum* that reflects the *degree of sharing* between processes. The full spectrum includes many different styles of message passing, as well as monitors, path expressions, remote procedure calls, atomic and parallel data structures, and unconstrained shared memory. In a pure shared-memory approach, processes share everything; in a pure message-passing approach, they share nothing. The other options lie somewhere in-between.

The continuum has not been widely recognized. Parallel programming environments have tended to present a single user view, often one directly supported by the underlying hardware. But a kernel interface is more than just a mechanism for accessing physical resources. It is also a programming abstraction that profoundly influences the algorithms that can be implemented on top of it.

Three years ago, our department acquired a 128-node BBN Butterfly^{TM} Parallel Processor [9], still the largest shared-memory machine available, and one that also provides firmware support for message passing. Since then, a major thrust of our work has been the comparison of solutions to common problems under various programming models [12, 22, 23, 24]. We are convinced that no one model of parallelism will prove appropriate for all applications. Some algorithms will be easier to implement with fully shared memory. Others are most clearly conceived with message passing. Still others need an intermediate option, such as monitors. Some applications may even benefit from the ability to use different models in different software modules. A computer vision system, for example, may be easiest to construct with shared memory at the lowest levels, where processes are operating in parallel on common pixel maps, and message passing at higher levels, where the emphasis is on feature integration in order to recognize objects.

The need for flexibility in the communication structures of parallel programs is illustrated by an analogy to the information structures of sequential programs. In sequential programming, information can be made available in one of two forms: a data structure that contains the information or a function that computes it. Since either approach can be used to implement the other, the choice depends on the attributes of the application. Information that is hard to compute, but easy to store and access, is encoded in a data structure. A data structure might also be used in situations where the relationship between data items, as encoded in the data structure, may be difficult to recreate. Information that is easy to compute, or would require too much space to store, is encoded in a function. Complex information structures, such as the symbol table in a compiler, often use combinations of both mechanisms.

Message passing is analogous to information exchange via functions, in that both impose a *value-oriented* semantics. Processes may only communicate values, some of which might require the exchange of an environment in which to interpret the value. The implicit communication required to establish an environment will often dominate the cost of interpreting a value within the environment. In the case of functions, a value-oriented semantics guarantees the absence of side-effects, but requires the environment to be passed as a parameter.[1] As with message passing, the cost of passing the environment as a parameter can dominate the cost of function execution.

Another property shared by message passing and functions is that both offer a form of abstraction. A function computes a value without requiring the caller to know any details of how the value is computed. Similarly, message passing offers a recipient the contents of a message without requiring it to know the details of how the message values were computed, when the message was sent, or what buffering operations were involved.

On the other hand, communication using shared memory is analogous to information exchange via data structures. Each computation (process) has access to the results of previous computations that have been stored (cached) in the shared memory, just as each procedure may have access to previous results stored in global data structures. Computation units (processes or procedures) have reduced fixed overhead, since they can inherit a context implicitly (an address space or a global data structure). There is little abstraction involved since both shared memory and data structure access require the user to have detailed knowledge of the location and format of information.

The analogy between communication structures and information structures is useful because it points out the inadvisability of any attempt to impose a single model of communication on all applications. Sequential programming systems do not attempt to dictate the choice of information structure; they provide functions, data structures, and hybrid combinations. Existing parallel programming systems tend to allow only a single communication structure. Psyche is designed to be more flexible, providing shared memory, message passing, and options in-between.

Lightweight Process Models

The processes scheduled by an operating system tend to be bulky objects with a large amount of state. Context switching between them is relatively expensive. Though many parallel algorithms are most easily realized with a very large number of processes, the cost of heavyweight context switches (as well as the space required for process state) makes straightforward implementation impossible. Lightweight processes, with a limited amount of explicit state, have been provided by several operating systems, including Mach [1] and Amoeba [27], and by an even larger number of parallel programming languages and library packages. The precise semantics of lightweight processes, however, differ nearly as much from system to system as do the semantics of interprocess communication.

As with IPC semantics, we believe that the choice of a lightweight process model must be left to the writers of individual applications. Certainly an operating system that intends to allow the implementation of LISP futures [18], Ada tasks [35], LYNX threads [32], Emerald objects [11], Modula-2 coroutines [37], and SR [4] processes cannot insist on the use of a single, fixed model for lightweight process management. Psyche provides a notion of thread that is independent of process weight, and that eliminates the need for kernel intervention

[1] We are assuming pure functions that do not have access to an implicit environment. Functions that reference global data are considered a hybrid form of information structure.

when switching between mutually-trusting threads.

Psyche Overview

The design of Psyche is based on the observation that access to shared memory is the fundamental mechanism for interaction between threads of control on a multiprocessor. Any other abstraction that can be provided on the machine must be built from this basic mechanism. An operating system whose kernel interface is based on direct use of shared memory will thus in some sense be universal.

Basic Concepts

The **realm** is the central abstraction provided by the Psyche kernel. Each realm includes data and code. The code constitutes a protocol for manipulating the data and for scheduling threads of control. The intent is that the data should not be accessed except by obeying the protocol. In effect, a realm is an abstract data object. Its protocol consists of operations on the data that define the nature of the abstraction. Invocation of these operations is the principal mechanism for communication between parallel threads of control.

The **thread** is the abstraction for control flow and scheduling. All threads that begin execution in the same realm reside in a single **protection domain**. That domain enjoys access to the original realm and any other realms for which access rights have been demonstrated to the kernel. The layout of a thread context block is defined by the kernel, but threads themselves are created and scheduled by the user. The kernel time-slices on each processor between protection domains in which threads are active, providing upcalls [15] at quantum boundaries and whenever else a scheduling decision is required.

The relationship between realms and threads is somewhat unusual: the conventional notion of an anthropomorphic process has no analog in Psyche. Realms are passive objects, but their code controls all execution. Threads merely animate the code; they have no "volition" of their own.

Depending on the degree of protection desired, an invocation of a realm operation can be as fast as an ordinary procedure call or as slow as a heavyweight process switch. We call the inexpensive version an *optimized* invocation; the safer version is a *protected* invocation. In the case of a trivial protocol or truly minimal protection, Psyche also permits direct external access to the data of a realm. One can think of direct access as a mechanism for in-line expansion of realm operations. By mixing the use of protected, optimized, and in-line invocations, the programmer can obtain (and pay for) as much or as little protection as desired.

Keys and **access lists** are the mechanisms used to implement protection. Each realm includes an access list consisting of <key, right> pairs. The right to invoke an operation of a realm is conferred by possession of a key for which appropriate permissions appear in the realm's access list. A key is a large uninterpreted value affording probabilistic protection. The creation and distribution of keys and the management of access lists are all under user control, enabling the implementation of many different protection policies.

Memory Model

If optimized (particularly in-line) invocations are to proceed quickly, they must avoid modification of memory maps. Every realm visible to a given thread must therefore occupy a different location from the point of view of that thread. In addition, if pointers are to be stored in realms, then every realm visible to multiple threads must occupy the same location from the point of view of each of those threads. Satisfying these two conditions simultaneously constitutes an exercise in bipartite graph coloring. In order to accommodate arbitrary changes to the graph at run time, we must generally arrange for all coexistent realms to occupy disjoint virtual addresses. Psyche therefore presents its users (conceptually at least) with a single, global, virtual address space. Each protection domain may have a different **view** of this address space, in the sense that different subsets may be marked accessible, but the mapping from virtual to physical addresses will be uniform. Virtual addresses suffice for naming, and pointers can (with appropriate permissions) be used without regard to the realm into which they point.

The view of a protection domain is embodied in the hardware memory map. Execution proceeds unimpeded until an attempt is made to access something not included in the view. The resulting protection fault is fielded by the kernel, whose job it is to either (1) announce an error, (2) update the current view and restart the faulting instruction, or (3) perform an upcall into the protection domain associated with the target realm, in order to create a new thread to perform the attempted operation. In effect, Psyche uses conventional memory-management hardware as a cache for software-managed protection. Case (2) corresponds to optimized invocation. Future invocations of the same realm from the same protection domain will proceed without kernel intervention. Case (3) corresponds to protected invocations. The choice between cases is controlled by the keys and access lists.

The major disadvantage of the uniform virtual address space is that address bits will be a scarce resource on most current architectures. Neither the 24-bit virtual addresses of many current machines nor the 32-bit virtual addresses now becoming available will be sufficient to address every realm of every program. Therefore, although the conceptual model provided by Psyche is that of a single, uniform address space, any practical implementation must take special measures to economize on virtual addresses. As with all scarce resources, it becomes important to (1) multiplex the resource among different programs and (2) reclaim the resource when it is not in use.

A Psyche implementation need only maintain the *appearance* of a uniform virtual address space. It can multiplex addresses if it knows that certain realms will never be simultaneously visible. Realms that will not be shared at all can clearly overlap. Substantial amounts of code and data are likely to fall into this category in practice. Asking the user to identify unshared realms to the kernel runs counter to the Psyche philosophy, but is likely to produce benefits that outweigh its conceptual cost. In addition, realms that are accessed only through protected invocations can be located somewhere other than where the user thinks they are, and in fact can overlap. Since the kernel is involved in every invocation, it can map a dense range of virtual addresses onto the operations of the overlapped realms.

In order to reuse virtual addresses, a kernel implementation must be able to tell when a realm is no longer needed. Since we want to support long-term sharing relationships, we cannot delete a realm simply because no thread is currently accessing it. Genuine garbage collection is also impractical, since it presupposes that all references to realms can be found. Other solutions adopted in traditional operating systems don't work either because sharing must be established explicitly beforehand, because long-lived sharing relationships are not allowed, or because resources that can be shared long-term are never reclaimed by the system. For example, in most operating systems memory is reclaimed when a process terminates. The file system must be used for long-term sharing (often defined to be any sharing that spans process boundaries) and file space is reclaimed only by human intervention. In Psyche, we plan to use a combination of explicit deallocation by the user and implicit deallocation via an ownership hierarchy to reclaim virtual address space. Explicit deallocation allows the user to micro-manage the virtual address space; implicit deallocation based on ownership guarantees that the system has ultimate control over resource reclamation.

Threads and Scheduling

Each realm in Psyche is the root of exactly one protection domain. All threads that begin execution in the same realm belong to the protection domain rooted in that realm. They share a common view of memory, that is, a single memory map. Initially, the view includes only the data of the original realm of the protection domain. When an attempt is first made to access another realm from that domain, the kernel checks access rights and implicitly *opens* the new realm for access by threads in the domain.[2] If optimized access is permitted, the new realm is added to the view.

The kernel time-slices on each processor between protection domains in which threads are active, providing each with an equal percentage of the CPU. On a given processor, each protection domain will be represented by at most one of its threads at any point in time. The identity of this thread can be changed in user code, so that the thread suspended at the end of a quantum may well be different from the one that was resumed at the beginning of the quantum. In effect, the kernel and user schedule exactly the same abstraction.

Each realm is required to provide routines for thread management tasks that involve the kernel. The kernel performs upcalls to these routines whenever user-level scheduling may be required. For example, upcalls occur when (1) an invocation of one of the realm's operations has occurred in a protection domain in which the realm is open for protected access (so that it may be appropriate to create a new thread to perform the requested operation), (2) a protected invocation by the current thread in the realm's own protection domain has caused that thread to block (so that it may be appropriate to run a different thread), (3) a protected invocation has completed in some other protection domain (so that a local thread may be unblocked), (4) a user-specified time limit has expired (so that preemption of the current thread may be required), and (5) a hardware fault has occurred (so that it may be appropriate to raise an exception in the current thread). None of these upcalls is expected to return. The state of the machine at the time of the upcall is saved by the kernel in the context block of the current thread. After performing its scheduling operation, the upcall routine is expected to jump immediately into the execution of an appropriate thread.

Upcalls execute in user mode, running code provided by the user. Their work space is allocated out of a static area established by the kernel when the realm is created. Each realm exercises complete control over the threads in its own protection domain. The kernel makes no assumption about the nature (or even the existence) of stacks for the threads themselves.

Since a realm can be opened for optimized access from more than one protection domain, it is possible for threads of many different kinds to be executing in the realm at once. In order to facilitate synchronization of these threads, each root realm of a protection domain is expected to provide a pair of routines to be called in user mode to block the current thread and to unblock a specified thread. These routines are in addition to the kernel-required upcall interface. When execution of a realm operation cannot proceed because of a synchronization constraint, the approved course of action is to call the thread blocking routine of the current protection domain, after saving the address of the unblock routine in an appropriate data structure. Low-level, architecture-specific primitives (such as test-and-set or compare-and-swap instructions) can be used to maintain atomicity of the scheduling operations.

In comparison to the library-based coroutine packages of traditional operating systems, the parameterized thread management of Psyche allows a protection domain to schedule other threads when the current thread has blocked, and permits time-slicing between user threads in a completely natural way. In comparison to the kernel-supported threads of Mach [1], or Amoeba [27], the Psyche mechanism provides the speed of a coroutine package for voluntary context switches within a protection domain and, given sufficient overlap of domains, for unblock operations that span thread types. In addition, the Psyche mechanism allows us to use the syntax and linkage conventions of ordinary procedure calls for both protected and optimized invocations. Once a realm is opened, it allows the optimized invocations to exhibit the same performance as ordinary procedure calls. Finally, the Psyche mechanism provides a much higher degree of flexibility than is possible with either other approach. Reference parameters can be used for protected invocations if the caller trusts the callee. Synchronization of operations in shared realms can be provided to dissimilar threads. User specification of the code to be executed by upcalls means that a realm can implement an explicitly message-based style of serving external requests, dispatching invocations to waiting server threads rather than creating new threads implicitly.

Keys and Access Lists

From the caller's point of view, protected and optimized calls will usually look the same. The exception is that a caller can insist that an invocation be protected when it does not trust the realm it is calling. In effect, Psyche has separated the dimensions of protection and performance from the semantics of realm invocation. Unless explicitly requested by the caller, the choice between the two is based on the access list of the realm being called.

When a thread attempts to invoke an operation of a realm for the first time, the kernel performs an implicit *open* operation on behalf of the protection domain in which the thread is executing. In order to verify access rights, the kernel checks to see whether the thread possesses a key that appears in the realm's access list with a right that would permit the attempted operation. Once a realm has been opened from a given protection domain, access checks are *not* performed for individual realm invocations, even those that are protected (and hence effected by the kernel).

Rights contained in access lists include; initialize realm (change protocol), destroy realm, invoke protected, invoke optimized (or in-line), and invoke optimized read-only.

Since the value of a key depends on neither the holder nor on the realm(s) to which it confers rights, it is possible to (1) possess a key that grants rights to a large number of realms, (2) change the rights conferred by a key without notifying the holder(s) and (3) change the holders of a key without notifying the realm(s) to which the key grants access.

The context block of each thread contains a pointer to the key list to be used when checking access rights. When a fault occurs, the kernel matches the key list of the current thread against the access list of the target realm. The principal drawback of this strategy is the potential cost of matching when both the key list and the access list are long. Since matching occurs only when realms are opened, there is reason to believe that any cost incurred will be amortized over enough operations to make it essentially negligible. Moreover, we believe that most programmers will use keys in either a capability or access-list style, so that either the key list or the access list will generally be short. In cases where multi-way matching is expected to be unacceptably slow, programmers will have the option of calling

[2] This "lazy evaluation" approach to protection frees the programmer from keeping track of which realms have been opened, and allows us to limit the cost of access rights verification to cases in which the realm in question will actually be used.

an explicit open operation, with explicit presentation of a key.

In the early stages of our design work, before adopting our system of keys, we had planned to use capabilities for protection in Psyche. This seemed to be a reasonable choice; realm invocations bore a superficial resemblance to mechanisms employing capabilities in several other systems, including the object invocations of Eden [3] and the procedure calls of Hydra [38]. Upon further examination, however, it became clear that the use of capabilities in Psyche would pose several serious problems:

(1) The tight association between names and rights within a capability would require most pointers into realms to be accompanied in every data structure by an appropriate capability, resulting in unacceptable space overhead.

(2) Given appropriate rights, our goal for optimized access is to map a realm into the current address space in such a way that further proof of rights is never needed. Under these circumstances we expect accesses to occur frequently enough to make the cost of presenting a capability on every access unacceptable, even if no actual verification is performed.

(3) Mandatory use of an explicit *open* primitive would eliminate the need to present capabilities for routine access, but would also force the Psyche user to keep track of which realms are currently accessible. Our experience with the Chrysalis operating system [7] has convinced us that this burden will be unacceptable for ordinary programmers and undesirable for the implementors of communication models. Opening realms at the earliest possible moment (rather than waiting until just before the first access) is also unattractive, because the set of realms that might potentially be accessed is likely to be very much larger than the set that will actually be accessed.

Traditional access lists solve these problems, but have other limitations of their own. They require that we be able to name the entities to whom access should be granted. They can require a great deal of space to list all valid names. They make it difficult or impossible to pass rights on to a third party without kernel intervention. By introducing keys as an additional level of indirection, we obtain the advantages of access lists while avoiding their disadvantages. Keys can be moved from place to place without kernel intervention. A single key can convey an arbitrarily complex set of rights over an arbitrary set of realms to an arbitrary set of clients. The rights associated with a key can even be changed without the knowledge of the clients. While it is not in general possible to prevent a thread from passing its keys on to a third party, we see no way to avoid this problem in any scheme that transfers rights between protection domains without the help of the kernel.

Locality

The fact that Psyche is intended to run on a large-scale multiprocessor raises locality issues not encountered in uniprocessors or in bus-based multiprocessors. Machines that will scale to hundreds or thousands of nodes must clearly have NUMA (non-uniform memory access) architectures. Current designs include the BBN Butterfly [8,9], IBM RP3 [29], Illinois Cedar [20], and Encore Ultramax [36]. Given hardware or firmware support for microsecond access to remote memory, hypercube designs would qualify as well. Optimal performance on these NUMA machines depends critically on maximizing locality, so that data accessed frequently is also accessed quickly. Unfortunately, the research community has yet to develop any general purpose memory management strategy that achieves the desired result. Attacking this so-called "NUMA problem" will be a crucial task for Psyche.

Psyche realms provide a strong notion of locality in our current implementation. All the data of a given realm resides at a single location, equally close or equally far from each individual thread. Applications that need to manage locality explicitly can create multiple realms. Allowing the data of a realm to be scattered across the machine would require either (1) a successful solution to the NUMA problem (in the form of kernel-managed, automatic, optimal data distribution), or (2) the introduction of a new abstraction to represent the pieces of a realm. We are reluctant to accept the latter; we do not yet have the former. We see our current approach as a reasonable first cut that will permit further experimentation.

Whether realms span NUMA boundaries or not, protection domains clearly must do so, since they consist of multiple realms. As a result, interactions between realms may span NUMA boundaries. In most cases performance will be maximized by executing realm operations on a processor close to the data. These operations must be performed by a thread co-located with the data. In some cases, however, the cost of transferring control to a thread on an appropriate processor may exceed the cost of accessing the data remotely. If the appropriate code is replicated, these operations can be performed by any thread, which then accesses the data remotely.

In our current implementation, we permit the user to specify which operations are data-intensive enough to justify the cost of co-locating code and data. The code for these operations is kept out of the page table to force the use of protected invocations, thereby transferring control to a thread in a domain that is close to the data. As with scattering of data, we consider this approach to be a first cut that will support later experimentation with more sophisticated realm or thread migration strategies.

The opportunity to perform migration occurs in several places. When a realm is first opened for optimized access from a given protection domain, the kernel can consider moving the realm to be closer to other realms in the domain. When a protected invocation provides reference parameters, the kernel can consider moving either the target realm or the realm of the parameters in order to optimize access. When a very large data structure is incorporated in the views of more than one protection domain, the kernel may use virtual memory techniques to copy on reference or even move on reference. The optimal mix of techniques for automatic data (re)location is far from obvious; the NUMA problem is very much an open research issue.

Although Psyche is not designed explicitly for loosely coupled networks, it could be extended to accommodate them in at least two different ways. The first is to incorporate networks into the NUMA model by simulating remote operations in software (i.e. in the kernel). This approach has the considerable advantage of functional transparency. Protected invocations would be implemented in much the same way as on a shared-memory machine. Optimized invocations would need to make use of automatic migration in order to obtain acceptable performance. Work by Li and Hudak suggests that this first approach is tractable for certain usage patterns [25]. In other cases, however, it might serve to hide costs that would be better kept explicit. An alternative would be to write network interface realms to support cross-machine operations. This second approach would require no kernel modifications. It is similar in style to remote procedure call stub generators and to the network server processes of Accent and Mach.

Examples of the Use of Realms

For both locality and communication, the philosophy of Psyche is to provide a fundamental, low-level mechanism from which a wide variety of higher-level facilities can be built. Realms, with directly-executed operations, can be used to implement the following:

(1) Pure shared memory in the style of the BBN Uniform System [10]. A single large collection of realms would be shared by all threads. The access protocol, in an abstract sense, would permit unrestricted reads and writes of

individual memory cells.

(2) Packet-switched message passing. Each message would be a separate realm. To send a message one would make the realm accessible to the receiver and inaccessible to the sender.

(3) Circuit-switched message passing, in the style of Accent [30], Charlotte [5], or Lynx [32]. Each communication channel would be realized as a realm accessible to a limited number of threads, and would contain buffers manipulated by protocol operations.

(4) Synchronization mechanisms such as monitors, locks, and path expressions. Each of these can be written once as a library routine that is instantiated as a realm by each abstraction that needs it.

(5) Parallel data structures. Special-purpose locking could be implemented in a collection of realms scattered across the nodes of the machine, in order to reduce contention [16, 17]. For certain kinds of data structures, (the Linda tuple space [2], for example), the entry routines of the data structure as a whole might be fully parallel, able to be executed without synchronization until access is required to particular pieces of the data.

Machine Requirements

In order to support an implementation of Psyche, a target multiprocessor must have certain characteristics. All or most of its memory must be sharable – the architecture may be UMA or NUMA, but it must be possible to access the code and data of any realm from any processor. The virtual address space must be at least as large as, and preferably much larger than, the physical address space. There must be a very large number of individually protected segments or pages. Support must be provided for very sparse address spaces.

Commercial multiprocessors that are likely candidates for Psyche implementations include the Sequent Balance, Encore Multimax, multiprocessor VAX, and BBN Butterfly machines. Of these, the Butterfly has by far the largest number of processing nodes and the most interesting memory architecture, in terms of varying locality. The new Butterfly 1000 series [8] also provides 32 bit virtual addresses, more than either Sequent or Encore.[3] Even with 32 bits, however, software techniques for coping with the scarcity of virtual addresses will still be necessary. Our implementation effort has deliberately focused on issues that are independent of the choice of a particular target machine.

Relationship to Previous Work

Psyche resembles Hydra [38] in its use of protected procedure calls for the execution of operations in separate protection domains. Our approach differs in its emphasis on multiple programming models, its integration of code and data in realms, and its provision for optimized access. Objects in Hydra can be either procedures or data. Realms in Psyche are both. Our approach is more in keeping with current use of the term "object-oriented," in that data is never separated from the protocol for its access.[4] Sharable data in Hydra can be accessed only through the use of capabilities, so very fine-grain operations, even without the need for protection, cannot be made efficient.

[3] The original Butterfly, the Sequent Balance, and the Encore Multimax all employ 24-bit virtual addresses, enough to access 16 megabytes. A fully-configured, 256-node Butterfly would contain one gigabyte of physical memory. The Balance can have up to 28 megabytes of memory, the Multimax up to 128 megabytes.

[4] The fundamentally passive nature of a realm, the unusual protection mechanism, and the lack of inheritance lead us to avoid the adjective "object-oriented."

The structural difference between Hydra objects and Psyche realms is best viewed as a difference in approaches to building abstractions. The association between data and procedures in Hydra is established by convention. Protocols are enforced by giving a procedure the ability to *amplify* the rights of capabilities for certain types of data objects. User programs hold capabilities that do not permit them to access the internals of the data objects; only the amplifying procedures can do so. Psyche abstractions, by contrast, are provided directly by the Psyche kernel. No amplification mechanism is needed in order to enforce the use of protocols. Where a Hydra user would ask the "pop" procedure to return an item from stack object **X**, a Psyche user would ask the "stack **X**" object to pop itself and return the result. By analogy to programming languages, the Hydra approach to abstraction resembles an Ada package [35] that exports an opaque type, while Psyche abstractions resemble Smalltalk objects.

Psyche also bears a resemblance to the StarOS [19] and Medusa [28] operating systems for Cm*. It is closer to Hydra than to StarOS, and closer to StarOS than to Medusa. StarOS emphasizes the asynchronous execution of operations by remote processes. As in Hydra, code and data comprise separate objects, but a number of special object types (dequeues, mailboxes, events) are built into the kernel and supported with microcode. A mechanism is provided for mapping an object into one of a limited number of *windows*, but the result is much less general than the inclusion of Psyche realms in views. In any event the use of a uniform virtual address space would not have been an option on the Cm* hardware, which only supported 16-bit addresses. Medusa adopts an essentially message-based approach to process interaction, with only a limited form of data sharing permitted within multi-process task forces.

Perhaps the best-known current work in multiprocessor operating systems is the Mach project [1], again at CMU. In comparison to Mach, Psyche has both a different motivating philosophy and a different set of resulting abstractions. Psyche is not constrained to be UNIX compatible. It is also not designed specifically for networks, though it could be extended to run in a loosely-coupled world. Its real focus is on scalable shared-memory multiprocessors, for which we believe it can make significantly better use of the hardware than is possible with a primarily message-based system.

Psyche adopts a passive view of objects, as opposed to the active view of Mach. Where Mach provides messages as the basic communication mechanism, Psyche provides data sharing and protected procedure calls. Where the notion of threads within a task is built into Mach at the kernel level, the threads of Psyche can be scheduled in user code and can move between mutually-accessible realms. Where Mach supports data sharing primarily between related tasks in the task creation tree, Psyche facilitates dynamic sharing relationships between arbitrary threads. Where Mach relies on the kernel to control the use of capabilities, Psyche provides probabilistic protection with keys in user space. All of these differences make Psyche a lower-level, less structured operating system, but at the same time one that will admit a wider variety of user applications with a finer grain of interaction.

We feel that the closest parallels to Psyche can be found in the so-called *open* operating systems developed for uniprocessors by groups at Xerox and MIT. In Cedar [34] (no relation to the Illinois Cedar project) and Swift [15], all the software of the machine runs in a single address space, with no protection provided by the kernel. Processes are prevented from interfering with each other by relying on the compiler for a "safe" programming language. Psyche can be regarded as an attempt to provide the advantages of an open operating system without relying on a single programming language. It is also an attempt to extend support to multiple processing nodes, though the Cedar group is moving in the same direction [6].

The comparison to Swift is particularly apt. The multi-process modules of Swift are very much like realms. Upcalls between modules in Swift resemble optimized realm invocations. Both Psyche and Swift are designed to separate the crossing of functional boundaries (i.e. between realms) from the expense of context switching. The solution may be more successful in Swift, since the CLU compiler can provide cost-free protection when calling an untrusted module. Psyche invocations that go "down" into a trusted realm like the file system will be easier to optimize than invocations that go "up" into untrusted user code.

Status and Plans

Design of the low-level kernel routines for Psyche was completed in the summer of 1987. Implementation of these routines has proceeded in parallel with the design of higher layers. We have recently acquired a 24-node Butterfly 1000 Parallel Processor (a.k.a. Butterfly Plus) on which we are continuing development. With its Motorola 68851-based memory management system, this new machine permits the large sparse address spaces we require. Our principal goal for the coming year is to obtain an environment as quickly as possible in which we can experiment with multi-model programs.

We expect our work to evolve into a number of interrelated projects. Interesting research could be performed in memory management (particularly for the automatic management of memory with non-uniform access times), lightweight process structure, implementation and evaluation of communication models, and parallel language design. The latter subject is of particular interest. We have specifically avoided language dependencies in the design of the Psyche kernel. It is our intent that many languages, with widely differing process and communication models, be able to coexist and cooperate on a Psyche machine. We are interested, however, in the extent to which the Psyche philosophy itself can be embodied in a programming language.

The communications facilities of a language enjoy considerable advantages over a simple subroutine library. They can be integrated with the naming and type structure of the language. They can employ alternative syntax. They can make use of implicit context. They can produce language-level exceptions. For us the question is:. to what extent can these advantages be provided without insisting on a single communication model at language-design time? Though these questions are beyond the scope of our current work, we expect them to form the basis of a future, follow-on project.

Acknowledgments

Many members of the Rochester systems group have contributed to the work reported herein. The authors extend their thanks to Rob Fowler, Bill Bolosky, Alan Cox, Lawrence Crowl, Peter Dibble, Neal Gafter, John Kerber, and John Mellor-Crummey.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986, pp. 93-112.

[2] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *Computer 19*:8 (August 1986), pp. 26-34.

[3] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe, "The Eden System: A Technical Review," *IEEE Transactions on Software Engineering SE-11*:1 (January 1985), pp. 43-59.

[4] G. R. Andrews, R. A. Olsson, M. Coffin, I. J. P. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, "An Overview of the SR Language and Implementation," *ACM TOPLAS 10*:1 (January 1988), pp. 51-86.

[5] Y. Artsy, H.-Y. Chang, and R. Finkel, "Interprocess Communication in Charlotte," *IEEE Software 4*:1 (January 1987), pp. 22-28.

[6] R. R. Atkinson and E. M. McCreight, "The Dragon Processor," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, 5-8 October 1987, pp. 65-71.

[7] BBN Advanced Computers Incorporated, "Chrysalis® Programmers Manual, Version 3.0," Cambridge, MA, 28 April 1987.

[8] BBN Advanced Computers Incorporated, "Inside the Butterfly Plus," Cambridge, MA, 16 October 1987.

[9] BBN Laboratories, "Butterfly® Parallel Processor Overview," BBN Report #6149, Version 2, Cambridge, MA, 16 June 1986.

[10] BBN Laboratories, "The Uniform System Approach to Programming the Butterfly® Parallel Processor," BBN Report #6149, Version 2, Cambridge, MA, 16 June 1986.

[11] A. Black, N. Hutchinson, E. Jul, and H. Levy, "Object Structure in the Emerald System," *OOPSLA'86 Conference Proceedings*, 29 September - 2 October 1986, pp. 78-86. In *ACM SIGPLAN Notices 21*:11 (November 1986).

[12] C. M. Brown, R. J. Fowler, T. J. LeBlanc, M. L. Scott, M. Srinivas, and others, "DARPA Parallel Architecture Benchmark Study," BPR 13, Computer Science Department, University of Rochester, October 1986.

[13] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *ACM TOCS 4*:2 (May 1986), pp. 110-129. Originally presented at the *Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985.

[14] D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 129-140. In *ACM Operating Systems Review 17*:5.

[15] D. Clark, "The Structuring of Systems Using Upcalls," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985, pp. 171-180. In *ACM Operating Systems Review 19*:5.

[16] C. Ellis, "Concurrent Search and Insertion in 2-3 Trees," *Acta Informatica 14* (1980), pp. 63-86.

[17] C. Ellis, "Concurrent Search and Insertion in AVL Trees," *IEEE Transactions on Computers C-29*:9 (September 1980), pp. 811-817.

[18] R. H. Halstead, Jr., "Parallel Symbolic Computing," *Computer 19*:8 (August 1986), pp. 35-43.

[19] A. K. Jones, R. J. Chansler, Jr., I. Durham, K. Schwans, and S. R. Vegdahl, "StarOS, a Multiprocessor Operating System for the Support of Task Forces," *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, December 1979, pp. 117-127.

[20] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, "Parallel Supercomputing Today and the Cedar Approach," *Science* ***231*** (28 February 1986), pp. 967-974.

[21] T. J. LeBlanc, R. H. Gerber, and R. P. Cook, "The StarMod Distributed Programming Kernel," *Software – Practice and Experience* ***14***:12 (December 1984), pp. 1123-1139.

[22] T. J. LeBlanc, "Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study," *Proceedings of the* ***1986*** *International Conference on Parallel Processing*, 19-22 August 1986, pp. 463-466. Expanded version available as BPR 3, Computer Science Department, University of Rochester, January 1986.

[23] T. J. LeBlanc, "Problem Decomposition and Communication Tradeoffs in a Shared-Memory Multiprocessor," in *Numerical Algorithms for Modern Parallel Computer Architectures*, IMA Volumes in Mathematics and its Applications #16, Springer-Verlag, 1988.

[24] T. J. LeBlanc, M. L. Scott, and C. M. Brown, "Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor," *Proceedings of the ACM SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages, and Systems*, July 1988.

[25] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, 11-13 August 1986, pp. 229-239.

[26] H. H. Mashburn, "The C.mmp/Hydra Project: An Architectural Overview," pp. 350-370 (chapter 22) in *Computer Structures: Principles and Examples*, ed. D. P. Siewiorek, C. G. Bell, and A. Newell, McGraw-Hill, New York, 1982.

[27] S. J. Mullender and A. S. Tanenbaum, "The Design of a Capability-Based Distributed Operating System," *The Computer Journal* ***29***:4 (1986), pp. 289-299.

[28] J. D. Ousterhout, D. A. Scelza, and S. S. Pradeep, "Medusa: An Experiment in Distributed Operating System Structure," *CACM* ***23***:2 (February 1980), pp. 92-104.

[29] G. R. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the* ***1985*** *International Conference on Parallel Processing*, 20-23 August 1985, pp. 764-771.

[30] R. F. Rashid and G. G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, 14-16 December 1981, pp. 64-75. In *ACM Operating Systems Review* ***15***:5.

[31] M. L. Scott, "The Interface Between Distributed Operating System and High-Level Programming Language," *Proceedings of the* ***1986*** *International Conference on Parallel Processing*, 19-22 August 1986, pp. 242-249.

[32] M. L. Scott, "Language Support for Loosely-Coupled Distributed Programs," *IEEE Transactions on Software Engineering* ***SE-13***:1 (January 1987), pp. 88-103.

[33] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm* – A Modular Multi-Microprocessor," *Proceedings of the AFIPS* ***1977*** *NCC* ***46***, AFIPS Press (1977), pp. 637-644.

[34] D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann, "A Structural View of the Cedar Programming Environment," *ACM TOPLAS* ***8***:4 (October 1986), pp. 419-490.

[35] United States Department of Defense, "Reference Manual for the Ada® Programming Language," (ANSI/MIL-STD-1815A-1983), 17 February 1983. Available as Lecture Notes in Computer Science #106, Springer-Verlag, New York, 1981.

[36] A. W. Wilson, Jr., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Fourteenth Annual International Symposium on Computer Architecture*, 2-5 June 1987, pp. 244-252.

[37] N. Wirth, *Programming in Modula-2*, Third, Corrected Edition. Texts and Monographs in Computer Science, ed. D. Gries, Springer-Verlag, Berlin, 1985.

[38] W. A. Wulf, R. Levin, and S.P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York, 1981.